3-2017

Alternative Processor within Threshold: Flexible Scheduling on Heterogeneous Systems

Stavan Satish Karia
sk3870@rit.edu

Alternative Processor within Threshold: Flexible Scheduling on Heterogeneous Systems

by

Stavan Satish Karia

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Dr. Sonia Lopez Alarcon
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY
March 2017

Approved By:

Dr. Sonia Lopez Alarcon
Primary Advisor – R.I.T. Dept. of Computer Engineering

Dr. Amlan Ganguly
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Dr. Marcin Lukowiak
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Dedication

Dedicated to my parents Satish Somnath Karia, Bindu Karia

and my sister Stuti Karia.

Acknowledgements

I would like to express my gratitude to my advisor and mentor, Dr. Sonia Lopez Alarcon, for her invaluable guidance. Her confidence and faith in me, throughout my research, made me work harder and allowed me to grow professionally and personally. I am extremely grateful to Dr. Amlan Ganguly and Dr. Marcin Lukowiak for their constant guidance and for serving as my committee. I have no words to express my gratitude to my family for their endless love, concern, support and strength all these years and my friends for their continued encouragement.

Abstract

Computing systems have become increasingly heterogeneous contributing to higher performance and power efficiency. However, this is at the cost of increasing the overall complexity of designing such systems. One key challenge in the design of heterogeneous systems is the efficient scheduling of computational load. To address this challenge, this paper thoroughly analyzes state of the art scheduling policies and proposes a new dynamic scheduling heuristic: Alternative Processor within Threshold (APT). This heuristic uses a flexibility factor to attain efficient usage of the available hardware resources, taking advantage of the degree of heterogeneity of the system. In a GPU-CPU-FPGA system, tested on workloads with and without data dependencies, this approach improved overall execution time by 16% and 18% when compared to the second-best heuristic.

Table of Contents

List of Figures

List of Tables

Table 16. APT kernel allocation analyses for DFG Type-2 graphs.

Introduction

Modern applications in industry and research exhibit a substantial computational bound and the situation is gradually worsening. With these mainstream applications becoming intrinsically data hungry and computationally intensive; optimizations in all components; programming, system software and hardware have become very important. Many such applications require high performance and power efficiency, which is not achievable with traditional CPU (Central Processing Unit) systems or even cluster based supercomputers. In search for better alternatives, only after 2001, GPUs were used as co-processors for highly parallel computations operating on really large amount of data. GPUs (Graphic Processing Unit) now had transformed from just a graphics processing unit to a highly parallel general programming coprocessor. This idea of using GPUs for computations other than just graphics in CPU-GPU systems is referred to as GPGPU (General-Purpose computing on Graphics Processing Unit). Also unlike CPUs, FPGAs (Field-Programmable Gate Array) are able to ride the Moore's Law curve, continuing to provide more logic and memory resources with each new generation. Falcao *et al.* [1] concluded that the FPGA was faster for the smaller data size when the CPU, GPU, and FPGA implementations of a Low-Density Parity-Check decoder were compared. Other works by Fletcher *et al.* [2] and Llamocca *et al.* [3] strengthened the belief that different applications have different hardware requirements for best performance. Also, ASICs (Application-Specific Integrated Circuit) are used widely for specific applications to give better performances as compared to the generalized CPUs. Therefore, systems with different kinds of processors are becoming widely accepted for various types of

applications, ranging from object recognition to image analysis in SETI (Search for Extra Terrestrial Intelligence). Such systems are known as *heterogeneous systems*. A hardware system level diagram of a heterogeneous system with multiple CPUs, GPUs, FPGAs and ASICs is shown in Figure 1.

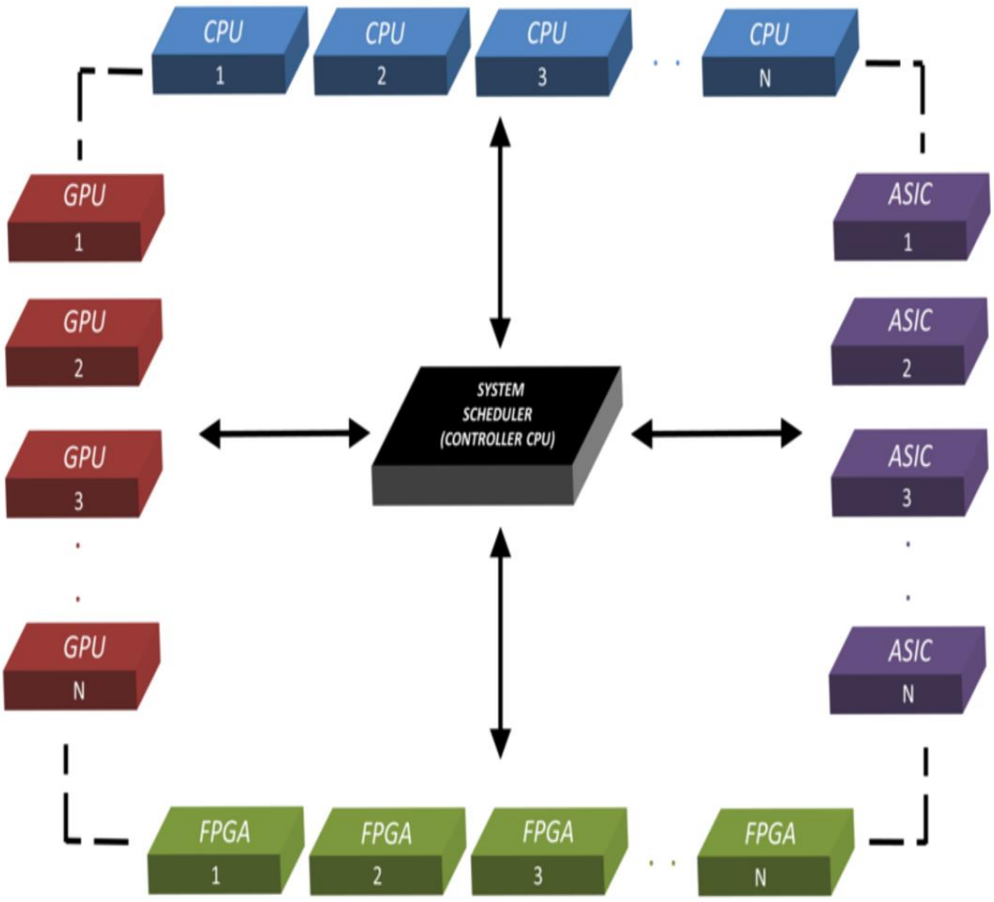

Figure 1. Hardware system level diagram of the heterogeneous system.

In section 1.1 we discuss the motivation behind using such heterogeneous systems for high performance computing applications and pose the problem statement that we will be addressing in this thesis. Later in section 1.2 we briefly describe our proposed solution

i.e. an optimized scheduling policy for heterogeneous systems with large degree of heterogeneity.

1.1. Motivation and Problem Statement

The limitations and design challenges associated with homogeneous systems; and comparative analysis of various applications on different kinds of processors have led to the emergence of heterogeneous systems. It has been understood that to achieve high performance and power efficiency, different kinds of applications have different hardware requirements. Binotto *et al*.[4] used a heterogeneous system of CPU, GPU and FPGA for X-ray image processing using high-speed scientific cameras. Also, Skalicky *et al.* [5] performed a distributed execution of transmural electrophysiological imaging with a heterogeneous system comprised of CPU, GPU and FPGA. It is shown in [4], [5] and many other works that using a heterogeneous system can give better performances in terms of total execution time, power efficiency and system utilization as compared to homogeneous systems.

There are many challenges of using heterogeneous systems which were presented by Khokhar *et al*. [6], such as, programming, hardware platform selection, best use of large degree of heterogeneity and network connections. Since then, many efforts have been made at simplifying programming for platforms like CPUs, GPUs and FPGAs. These include many libraries for CPUs, a variety of programming languages for GPUs and FPGAs in addition to cross-compilers and high-level synthesis tools. Connecting CPU, GPU and FPGA via PCI Express has been proposed by Chen *et al.* [7] and Skalicky *et al*. [8] to solve the problems of networking. Scheduling policies for mapping tasks from a directed acyclic

graph (DAG) to heterogeneous processors have been studied and found to be NP-complete for finding the optimal schedule [9]. Also, scheduling in heterogeneous systems has been heavily researched [11-14], but usually only with systems containing abstract hardware platforms. And in these studies, hardware platforms have been associated with generic heterogeneities rather than using specific hardware platforms. As the variety of real platforms included in current heterogeneous systems expands, the problem at hand is of finding the best scheduling heuristic for systems with high degrees of heterogeneity. Optimal assignment of work to hardware platforms is essential in achieving high performance and efficiency from heterogeneous systems.

1.2. Proposed Solution

In this work, after thorough analysis of the six state of the art scheduling policies for heterogeneous systems and comparing their performance for a variety of stream of applications, we propose an optimized scheduling heuristic for heterogeneous systems with high degree of heterogeneity. The six examined policies are, predict earliest finish time (PEFT) [15], heterogeneous earliest finish time (HEFT) [16], shortest process next (SPN), serial scheduling (SS) [17], adaptive greedy (AG) [18] and minimum execution time/best only (MET) [19]. In the proposed optimized heuristic, we consider a *tolerance threshold*, which is the deciding metric for an assignment of task to any processor. As opposed to the policies like SS, SPN and MET, this heuristic makes the decision to *wait* or to *assign to next best available processor* based on a metric which ensures that the total execution time is minimized and the system utilization is optimal. Also, this policy being dynamic, it does not need an intensive pre-computation phase like HEFT and PEFT. Also, unlike AG, which

capitalizes mainly on reducing communication time in the system, this policy tries to optimize the total execution time by capitalizing on the fact that there is abundance of multiple types of idle competing processors. This policy therefore strikes a good balance of efficiency and effectiveness.

Chapter 2 Related Work and Background

In this chapter we describe the previous work that forms the foundation of our work and other similar efforts that are related to our objectives and contributions. In section 2.1 we discuss the related work and in section 2.2 we elaborate on the advancements in heterogeneous computing. Section 2.3 represents the types of processors used in the heterogeneous systems with very large heterogeneity and the methodology of classifying *dwarfs* is presented in the section 2.4. Finally, in section 2.5 we present the scheduling problem and provide a survey of the state of the art scheduling policies for heterogeneous systems.

2.1. Related Work

Performance evaluations have compared and contrasted various computations and hardware platforms to determine which is best [20-22]. Skalicky et al. [23] evaluated five linear algebra computations using multiple implementations for each hardware platform. They presented the areas within the design space in which each processor architecture and implementation excelled. Their results represent the ground truth for making intelligent computation-to-hardware assignments to maximize performance. Also, Krommydas *et al*.[24] evaluated the performance of four different kinds of applications on different kinds of processors. The applications evaluated in [24] are Needleman Wunsch, GEM (Gaussian Electrostatic Model), BFS (Breadth First Search) and SRAD (Speckle Reducing Anisotropic Diffusion). We will use results from [23] and [24] to evaluate how well each scheduling policy mapped and assigned computations to hardware platforms.

As opposed to the previous work [11-14], we use specific hardware platforms in this work. Topcuoglu et al. [16] presented the highly regarded heterogeneous earliest finish time (HEFT) policy but do not mention the heterogeneity of their system. Arabnejad et al. [15] presented the predict earliest finish time (PEFT) policy that used a novel optimistic cost table and produced makespans of 20% less than HEFT using an abstract system where each platform had a heterogeneity value between 0 (similar) and 2 (very different). Liu et al. [17] presented the priority rule based serial scheduling (SS) policy and evaluated it in a system with uniformly distributed random task compute times. Wu et al. [18] presented the adaptive greedy (AG) algorithm and evaluated it in a heterogeneous system of CPU+GPU workstations but used exponentially distributed random task compute times. Braun et al. [19] presented eleven scheduling policies including opportunistic load balancing (OLB) and minimum execution time (MET) and evaluated them in a system with uniformly distributed random task compute times. However, OLB does not consider the execution time of each task on the given hardware platform before making assignments. The shortest process next (SPN) policy was suggested by Khokhar et al. [6] for use in heterogeneous systems and improves upon OLB by choosing the next task to assign based upon the shortest execution time of a task on any of the available hardware platforms.

2.2. Heterogeneous Computing

Until very recently, the most powerful HPC (High Performance Computing) systems were primarily CPU based [25], although there is a very recent but significant shift towards the use of general-purpose graphical processor unit (GPU) co-processing. On the other hand, many critics are of the opinion that the “Top 500” may not be representative of

the true compute power of a cluster [26]. Because power consumption, and hence heat generation, is proportional to clock speed, processors have begun to hit the so-called "*speed wall*". Meanwhile, hardware accelerators have occupied niches, such as video processing and high-speed DSP applications. The most commonly available of these accelerators are the (general-purpose) graphical processor unit (GPU) and the FPGA.

Systems that use more than one kind of processor are referred to as *heterogeneous computing*. These are multi-core systems that gain performance not by just adding more cores, but also by including specialized processing capabilities to handle specific tasks. There are a number of platforms that implement an on-chip or off-chip heterogeneous CPU+GPU+FPGA system. A sophisticated sixteen node cluster, known as the "Quadro-Plex Cluster" [27] is an example of such a heterogeneous system. It has two 2.4 GHz AMD Opteron CPUs, four nVidia Quadro FX5600 GPUs, and one Nallatech H101-PCIX FPGA in each node, with a thread management design matching that of the GPUs. Another such example is the "Axel" [28]. It is a configuration of sixteen nodes in a Non-uniform Node Uniform System (NNUS) cluster, each node comprising an AMD Phenom Quad-core CPU, an nVidia Tesla C1060, and a Xilinx Virtex-5 LX330 FPGA. Also, the "chimera" [29] is another example of a heterogeneous system of CPU, GPU and FPGA.

2.3. Types of Processors

Different kinds of processors have their own purpose in any system and therefore their own set of advantages and disadvantages. The general-purpose CPU is expected to perform a variety of tasks and therefore CPU processor designs cannot afford to specialize. These processors are built to have certain peculiar characteristics that make them suitable

for general almost all kinds of computational loads. CPUs are usually deeply pipelined, run at very high clock frequencies and have a lot of hardware on-board, etc. to run code speculatively/out of order. CPUs are very useful and perform the best when there is a lot control switching in the application, pointer usages, indirect load-stores etc.

Traditionally developed for graphics processing, GPUs today are used for almost any application which has lots of parallelism. This is because GPUs were designed to have a SIMD (Single Instruction Multiple Data) architecture with the vision to efficiently perform linear operations on vectors and matrices. Also, they use a lot less power when compared to CPUs for similar computations [30]. GPUs have hundreds or even thousands of stream processors, and each stream processor runs slow when compared to a CPU and also has less features; but collectively, the extremely high degree of parallelism in GPUs hide the latency and outperform CPUs in tasks with lots of parallelism.

An FPGA is very different from CPUs or GPUs in the sense that it is not a processor in itself i.e. it does not run a program stored in the program memory. FPGAs are special hardware implementations of specific algorithms/tasks and are more deterministic. Being special hardware implementations, they are faster than any software implementation. Also, they can be configured as needed and this makes them ideal for re-configurable computing and application specific processing. Another benefit of the custom design is that high performance can be achieved at lower frequency.

2.4. Dwarfs

The applications chosen for our work belong to multiple domains, ranging from gesture recognition and linear programming to molecular dynamic simulations. But we

know that each application can be broken down to a set of kernels and that each kernel in an application has a particular computational objective for which it follows a computation and communication pattern. This idea of breaking down an application into kernels where each kernel has a computational objective is illustrated in Figure 2.

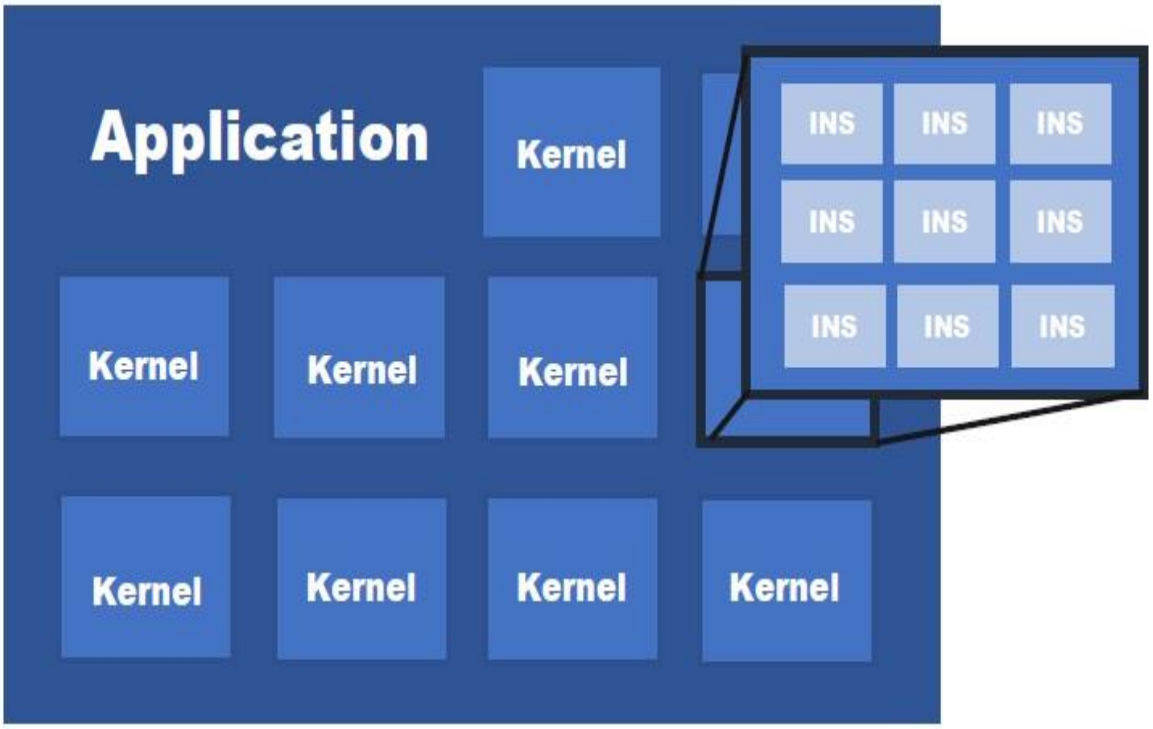

Figure 2. Application break down: an application has multiple kernels; each kernel has multiple instructions (INS).

An algorithmic method that captures a pattern of computation and communication is called a *dwarf*. In his work [31], P. Colella identified seven numerical methods that he believed will be important for science and engineering for at least the next decade. These

seven dwarfs can be understood as equivalence classes in which membership in a class is defined by the similarity in the computation and communication pattern i.e. data movement. Inspired from [31] and after exploring more applications, Asanovic *et al*. [32] expanded the list of dwarfs from seven to thirteen. Kernels that are members of a class can have different implementations and the core numerical methods may change too, but the underlying patterns have persisted for generations and will remain important in the future too. Below, with short descriptions, we list all the dwarfs presented in [32] and the dwarfs marked with * are the dwarfs that were newly introduced.

a) ***Dense Linear Algebra***: These are traditional vector and matrix operations, usually divided into three levels. The three levels are level 1 (vector/vector), level 2 (matrix/vector) and level 3 (matrix/matrix) operations.

b) ***Sparse Linear Algebra*** – Sparse matrices are the ones that have many zero entries. Sometimes, using another data structure has more advantages when it comes to memory and efficiency. Algorithms that involve such data structures and computations belong to sparse linear algebra category of dwarves.

c) ***Spectral Methods*** – These methods are widely used in many different fields like applied mathematics and scientific computing. In this method, data is operated on in the spectral domain, which is often transformed from a temporal or a spatial domain. Therefore, they often involve use of a Fast Fourier Transform (FFT).

d) ***N-Body Methods*** – These methods involve calculations that depend on interactions among many discrete points. This dwarf does not cover a few particle methods such as the particle-in-cell (PIC) codes.

e) ***Structured Grids*** – In this dwarf, data is formatted in a regular multidimensional grid. This grid is updated in a sequence of steps and in each step, the points are updated using values from its neighborhood.

f) ***Unstructured Grids*** – These methods are used when there are surfaces/objects or any modelling problem that has irregular geometric dimensions. When each grid element is updated, unlike structured grids, irregular number of neighboring elements are accessed, leading to an irregular amount of computations.

g) ***MapReduce*** – Initially, this dwarf was known as "Monte Carlo" after the idea of using statistical methods based on repeated random trials. Generalizing the same idea, this dwarf has the programming model in which a function is repeatedly executed independently and the results are aggregated at the end from all these independent executions.

h) ***Combinational Logic*** * - This dwarf has many important functions that exploit bit-level parallelism for high throughput. These dwarfs have computations in which the operations are quite simple logical operations, but they are operated on very large amounts of data.

i) ***Graph Traversal*** * - These are kernels which traverse a number of objects in a graph data structure while examining the characteristics of the objects, usually with very little computation.

j) ***Dynamic Programming*** * - Dynamic programming is a programming method in which a complex problem is solved by decomposition into smaller sub

problems. Subsequently, combining the solutions to the sub problems provides the solution to the original problem.

k) ***Backtrack and Branch-and-Bound*** * - These algorithms are very effective in solving search and optimization problems. The idea is to search for an objective in a very large space to find an optimal solution. Usually the search space is intractably large and a set of rules are devised to prune subregions of this search space that have no helpful solutions. This method uses the divide and conquer rule to divide the search space into smaller regions and then searches for solution in this smaller sub region.

l) ***Graphical Models*** * - These models are represented by a graph that has nodes which represent variables and edges that represent conditional probabilities. As these models are graphs, they are evaluated using graph traversal methods.

m) ***Finite State Machines*** * - This is a system that can be described as set of connected states. The behavior of such systems can be defined by states, transitions defined by inputs and the current state, and various events associated with transitions or states.

Understanding these dwarfs and the idea that applications consist of one or more kernels is key in identifying the dwarfs that are found in an application. This means, that an application can have kernels that belong to different kinds of dwarfs. For example, consider a Bayesian network model for a machine learning problem. This application has both, the graphical model dwarf which builds the model during the training of the system and the graph traversal dwarf that evaluates during the testing phase of the system. But there are applications that have only one kernel and just one dwarf. An example of this can

Table 1. Each column denotes the types of dwarfs and each row shows the belongingness of applications to dwarfs.

	Dense Linear Algebra	Sparse Linear Algebra	Spectral methods	N-Body methods	Structured Grids	Unstructured Grids	Graph Traversal	Dynamic Programming
Needleman Wunsch								■
Matrix Inverse	■							
GEM				■				
Cholesky decomp.	■							
BFS							■	
Mat.Mat. Multi.	■	■						
SRAD					■			
LavaMD				■				
HotSpot					■			
Backpropagation						■		
FFT			■					

be the BFS implementation for the shortest path problem which has just the Graph Traversal dwarf. Going forward with this idea, the Table 1. summarizes a variety of examples of applications. It also indicates all the dwarfs that are found in all these applications.

2.5. Scheduling

2.5.1 Problem Representation

We can represent the problem of scheduling kernels from an application in a heterogeneous system as ($R \mid prec \mid C_{max}$) in standard scheduling notation. For this problem, we have processors $p_j \in P$ for $1 \leq j \leq n_p$, where n_p is the number of processors in the system, and a dataflow graph $G = (V, E)$ where V is the set of kernels and E is the set of dependencies between kernels. Each kernel $v_i \in V$ has an execution time $t_{ij} \in T$ for processor j. For kernel v_i , the data transfer cost is $d_{jk} \in D$ when v_i's predecessor is assigned to processor p_j and v_i is assigned to p_k.

Mathematically a scheduling algorithm can be represented as a function f that maps kernels from V to processors in P as $f : V \rightarrow P$ such that each kernel is assigned to exactly one processor. Currently there are no feasible polynomial time algorithms to find the best kernel-to-processor assignment map, or schedule, by minimizing the maximum completion time of any kernel in the application. We do not find a lot of previous work focused on this problem in particular since the two relaxed simplifications still have no known polynomial time solutions. In essence, this work presents an approach that models the performance of heuristic solutions to this scheduling problem. Usually these scheduling policies are studied

statically, having access to the entire kernel dataflow graph (DFG) of the application. In the real world though, this may not be possible, so *dynamic scheduling* approaches are also used in large number of systems.

Foster [33], Puigjaner [34] and many others have been active researchers in modeling system performances. Ideally, the scheduling should be able to assign the kernels to the processor to achieve the lowest overall execution for the stream of applications. Since we cannot achieve the minimum execution time in real world, efforts have been made to identify the components of this overall execution time. We have understood that this overall execution time comprises of three parts: kernel compute time, data transfer time, and scheduling delay. The first two components, kernel compute time and data transfer time depend on the processors in the system and the system design. Therefore, we will only discuss the scheduling delay; the first two components being trivial. This delay, λ, as discussed earlier could be caused by various factors such as:

- the scheduling delay to process which task should be assigned to which processor next,
- communication delay from the scheduler to the processor to tell it to begin processing and provide the necessary information,
- dependencies on kernels that are being executed in another processor, but have not completed yet.

This means that the order in which tasks are assigned impacts the amount of scheduling delay. To understand this delay and its impact on the performance of scheduling policies, we compare the overall impact of this delay on the total execution time for each scheduling policy.

2.5.2 Types of Scheduling Policies

Elaborating a little more on types of scheduling policies, ***static scheduling*** policies have access to the entire DFG of the application prior to execution. This category therefore determines a schedule before executing the application on the heterogeneous system. The schedule that the policy gives beforehand, is followed during the actual execution. Some notable work in this category of scheduling policies was by Herrmann *et al*. [35] and Liu *et al*.[17]. Herrmann *et al*. investigated scheduling with a peculiar chain dependency structure and Liu *et al*. proposed a priority rule-based algorithm which had arbitrary dependencies. As compared to the policies mentioned before, ***dynamic scheduling*** policies do not have access to the entire DFG. These policies therefore try to make the best of the current state of the system and the kernels that have been already submitted. Adaptive Greedy and Adaptive Random were two policies presented in [18] by Wu *et al*. The Adaptive Greedy policy tries to minimize the waiting time for each kernel whereas the Adaptive Random policy uses random weights and probabilities to assign kernels. The system used for this investigation had multiple CPUs and GPUs. But in an approach like this, the kernels become resource constrained and need a different scheduling approach [36]. This is mainly because the kernels are custom and cannot be broken further down in a combination of standard library routines.

2.5.3 Chosen Scheduling Policies

In this work we analyze **two static** and **four dynamic** state of the art scheduling policies to assign kernels to processors. All of these policies assign kernels from a set of independent kernels to a set of available processors. The set of independent kernels, *I*, is a

subset of *V*. It is a set in which each kernel that has not yet begun execution and whose dependencies, also known as the precedence constraints, have already been completed. The set of available processors, *A*, is a subset of *P*. It contains only those processors that have are not currently executing any kernels or data transfers.

The ***shortest process next*** (SPN) policy was suggested by Khokhar *et al*. [37] chooses a kernel from *I* that has the minimum execution time on any of the processor from *A*. If there is any processor available and there are kernels in set *I*, assignments are made to keep the system busy. This policy tries to minimize λ delays by keeping the processor busy. But this policy has its own decision making mechanism, according to which it does not use the information about the difference in execution time among the processors. This mechanism therefore disregards the observed heterogeneity observed in the kernels, therefore not making the best use of available heterogeneity in the system architecture.

Braun *et al.* [19] presented the ***minimum execution time*** (MET) policy. In this policy, a kernel is chosen in a random order from *I* and is then assigned to the processor with the lowest execution time for that kernel. As opposed to SPN, if the best suited processor for the kernel is not currently available, policy decides to wait for the best processor to become available i.e. the kernel will be assigned to that best processor at a later time. By virtue of this rule, a processor sits idle if there are no kernels in *I* that are suitable for it. This policy always waits to assign kernels to their best processor. Due to the large differences in execution times, this will result in lower λ delays.

A relatively more statistical scheduling policy known as ***serial scheduling*** (SS) was presented by Liu *et al*. [17]. In this policy, the metric for decision making is the standard deviation of the compute times. To elaborate a little, for each kernel in *I*, the mean and

standard deviation of the compute times are calculated for each kernel-to-available-processor mapping. Then the scheduler chooses the kernel from *I* with the highest standard deviation and assigns it to the processor from *A* in which the kernel has the lowest execution time. Whenever there are kernels in *I* and there are available processors, assignments can be made in this policy. This policy is a little different than the policies mentioned previously. It does not directly consider the difference in execution time among the processors in its calculations; instead calculates the standard deviation in execution time among the processors and assigning kernels to the processor with the least execution time. When the best processor is busy, just like SPN, SS assigns kernels to processors even if they are the not the best choice.

The ***adaptive greedy*** (AG) policy presented by Wu *et al*. [18] tries to optimize the data transfer and queuing delay. It maintains queues for each processor and attempts to make assignments to minimize data transfer and queuing delay. The policy calculates wait time by adding the queuing delay for each processor and the associated data transfer time for the given data size and transfer rate. Then the policy chooses the processor which will incur the lowest total time. The queuing delay mentioned above is calculated as the sum of the compute times for all kernels already in the queue for each of the processors. This policy takes the differences in execution time between the various processors into account by using the queuing delay in its decision making metric. As it turns out, this policy indirectly ends up making the decision to wait for the best processor.

In [18], AG considers a CPU-GPU system, but we generalize the policy to a heterogeneous system with CPU, GPU and FPGA. This policy examines every device in the system and estimates the total waiting time τ_g in the case that the kernel is assigned to

the device g. As shown in (1), τ_g comprises of the queueing delay τ_g^q (time to queue the kernel to the processor g) and the data transfer delay τ_g^d (time to transfer the data that the kernel requires for successful execution). Also (2) explains that τ_g^q, the queueing delay is estimated by the number of kernel calls queued on that processor i.e. N_g and the average execution time of the last k kernel calls on that processor i.e. τ_g^k.

$$\tau_g = \tau_g^q + \tau_g^d \tag{1}$$

$$\tau_g^q = N_g . \tau_g^k \tag{2}$$

The ***heterogeneous earliest finish time*** (HEFT) policy presented by Topcuoglu *et al.* [16] is a static scheduling policy which makes its decisions based on a statistically derived rank. Because the policy is static, it has access to the entire kernel DFG beforehand, and using this DFG, the policy first statically ranks all kernels and then assigns them to processors in order of highest rank first in I. The assignments are made to the processor from A with the least sum of time remaining of any previous kernel and execution time of the current kernel on that processor. This policy was specifically designed to minimize the λ delays in the rank calculations by evaluating dependencies in the DFG.

Tasks in HEFT are ordered based on their scheduling priorities using their upward and downward rank. The *upward rank* of a task n_i is defined by (3), where *succ(n_i)* is the

$$rank_u(n_i) = \overline{w_i} + \max_{n_j \in succ(n_i)} \left(\overline{c_{i,j}} + rank_u(n_j) \right) \tag{3}$$

set of immediate successors of n_i, $\overline{c_{i,j}}$ is the average communication cost of edge (i, j), and $\overline{w_i}$ is the average computation cost of n_i. It is called the *upward rank* because it is computed recursively traversing the graph upward, starting from the exit task. The upward rank for the exit task is equal to

$$rank_u\,(n_{exit}) = \overline{w_{exit}} \tag{4}$$

Similarly, the downward rank is defined by

$$rank_d(n_i) = \max_{n_j \,\epsilon\, pred(n_i)} \left(rank_d\,(n_j) + \overline{w_j} + \overline{c_{j,i}}\right) \tag{5}$$

where *pred(n_i)* is the set of immediate predecessors of task n_i. It is called the *downward rank* because it is computed recursively traversing the graph downward, starting from the entry task of the graph. The downward rank value for the entry task n_{entry} is zero. From these definitions, we understand that the upward rank is the length of the critical path from n_i to the n_{exit}, including the computation cost of the task n_i. And the downward rank is the longest distance from n_{entry} to n_i, excluding the computation cost of the task itself. After selecting the task based on its calculated priority (using the upward and downward rank), the processor selection phase of HEFT is a little different than most other scheduling policies. It has an insertion-based policy which considers an insertion of task in an earliest time slot between two already scheduled tasks, if the time slot can accommodate the computation time of the chosen task.

The ***predict earliest finish time*** (PEFT) policy, yet another static policy, presented by Arabnejad *et al.* [15] follows a similar process to HEFT except that the ranks are based on a pre-computed cost table. This cost table serves as a lookup table that helps the policy in making decisions for allocation of kernel to the processor. The assignments are made to the processor from *A* with the least sum of value from the cost table and execution time of the kernel on that processor. Just like HEFT, this policy also specifically addresses λ delays in the rank calculations by evaluating dependencies in the DFG.

PEFT uses an optimistic cost table (OCT) based on which the task priority is decided and the processor is selected. The OCT is a matrix in which the rows indicate the number of tasks and the columns indicate the number of processors. Each element in the matrix $OCT(t_i, p_k)$ is the maximum of the shortest paths to t_i children's tasks to the exit node in the case that the task t_i is assigned to processor p_k. This value is defined by the formula shown in (6) by traversing the task graph from the exit task node to the entry task node. In (6), $\overline{c_{i,j}}$ is the average communication cost, $w(t_j, p_w)$ is the execution time of task t_j on processor p_w, $succ(t_i)$ is the set of immediate successors of t_i and P is the number of processors in the system. Also $\overline{c_{i,j}}$ is zero if t_j is being evaluated for processor p_k because

$$OCT(t_i, p_k) = \max_{t_j \epsilon\, succ(t_i)} \left[\min_{p_w \epsilon P} \{ OCT(t_j, p_w) + w(t_j, p_w) + \overline{c_{i,j}} \} \right], \quad \overline{c_{i,j}} = 0 \quad if \quad p_w = p_k \tag{6}$$

the task is going to be executed on the same processor and therefore there is no communication cost. And finally to make assignments, task priority is calculated using $rank_{oct}$ which is defined in (7). To select a processor for a task, O_{EFT} (Optimistic Earliest

$$rank_{oct}(t_i) = \frac{\sum_{k=1}^{P} OCT(t_i, p_k)}{P} \tag{7}$$

Finish Time) is calculated which sums to EFT the computation time of the longest path to the exit node. Comparing $rank_u$ and $rank_{oct}$, it is understood that $rank_u$ uses the average computing cost for each task and also accumulates the maximum descendent costs of descendent tasks to the exit node. In contrast, $rank_{oct}$ is an average over a set of values that were computed with the cost of each task on each processor.

A comparative analysis of the before mentioned policies can be found in Table 2. It summarizes the unique properties and decision metrics the policies adopt to assign tasks

to processors. Each column in the table represents a scheduling policy. This comparison helps in understanding the key differences in all these policies and the effects of these differences in the performance of these policies.

Table 2. Summary of key properties of the scheduling policies HEFT, PEFT, SS, AG, SPN and MET.

	HEFT	**PEFT**	**SS**	**AG**	**SPN**	**MET**
Scheduling policy Type	Static	Static	Dynamic	Dynamic	Dynamic	Dynamic
Heuristic/statistical rank calculation of kernels	Yes	Yes	Yes	No	No	No
Considers heterogeneity in execution times	Yes	Yes	Yes	Indirectly	No	Yes
Considers data transfer time	Yes	Yes	No	Yes	No	No
Never waits (if kernel and processor available)	Yes	Yes	Yes	No	Yes	No

From the before mentioned descriptions and Table 2, we understand that static policies i.e. HEFT, PEFT; have a ranking mechanism like a preprocessing step to prioritize the available set of subtasks in the application. Using the ranks from the ranking process, a fixed static schedule is formed and is followed during the execution. But for applications with high degree of parallelism and very deep DFG, the ranking step can be very time consuming and thus cumulatively very expensive. Among the dynamic policies, Serial Schedule (SS) is the only policy with a ranking process for kernels. But as discussed before,

this ranking is not iterative and neither is it as complicated as it is for the static policies. This policy tries to prioritize execution of kernels that have the maximum heterogeneity on a processor that is available and has the least execution time than the other available processors. While doing this, the policy might end up assigning kernels to a processor that is very expensive in terms of computation time. Shortest Process Next (SPN) has a simple rule for assignment i.e. assign the shortest available kernel to the best available processor. In this process, the policy does not consider the heterogeneity available in the system and therefore does not make the most effective use of the resources available. In both the policies, SS and SPN, the assignments are made at the cost of not caring about how slow the selected processor is as compared to most suitable processor which is currently unavailable. As opposed to SS and SPN, Minimum Execution Time (MET) has an even simpler approach of choosing the best processor for a kernel, whenever it is available, even at the cost of waiting time. This approach makes the best use of heterogeneity of the system, but will have large execution times when few processors are best at many different kernels, adding a lot of waiting time. When all other policies concentrate on reducing the execution time or execution time coupled with communication time, Adaptive Greedy (AG) tries to reduce waiting time for kernel execution and not computation time across processors. Therefore, the policy favors executing kernels on either the same processor or the processors connected with higher bandwidths. All the above mentioned dynamic policies have access to the observed heterogeneity in the kernels and multiple types of processors. But none of these policies capitalize on this heterogeneity among kernels and the availability of multiple computational resources at the same time.

Chapter 3 Alternative Processor within Threshold (APT)

In this chapter, section 3.1 describes the proposed scheduling heuristic, Alternative Processor within Threshold (APT) and in section 3.2 discusses the methodology to use the heterogeneous system and evaluate the seven scheduling policies (six state of the art examined policies and the proposed policy - APT).

3.1. Scheduling heuristic - Alternative Processor within Threshold (APT)

In this section, we introduce a new scheduling heuristic for heterogeneous systems, called Alternative Processor within Threshold. APT is a dynamic scheduling heuristic that adds flexibility to MET, a flexibility that can be tuned to the degree of heterogeneity of the system. This flexibility offered by APT, makes it more lenient in making the kernel-to-processor assignment instead of always waiting for the best suitable processor to be available. This policy has just one phase, the ***processor selection*** phase, to choose a processor which will execute the selected task.

Ideally, for a scheduling policy to be effective and reduce the overhead of scheduling delay, *the scheduling policy should be quick in choosing the task and the processor on which the task will be executed*. This means that, if the policy has lesser computations in selecting the task to be scheduled and a suitable processor, it reduces the λ delay, thus achieving better performance. But this improvement can come with the cost of longer schedules and higher overall execution times if the decision metric for this *quick*

assignment is not good enough. To address this issue, we try to keep the computations in the processor selection phase to a minimum in our policy.

APT maintains a list of tasks as and when they arrive for execution. This list can be referred to as a queue because it is filled on first-come, first-serve basis while maintaining the computational and data dependencies among different kernels. If there are tasks that are ready to be scheduled, each task in this queue gets a fair chance to find a suitable processor for execution. Once there are tasks that can be scheduled, the policy tries to find a suitable processor for the task in the ***processor selection*** phase.

For a chosen task v_i from the queue, the ***processor selection*** phase, tries to find a processor which has the minimum execution time for v_i. The minimum execution time can be found from the entries for v_i in the lookup table and we refer to the processor with the minimum execution time as p_{min}. The lookup table is a table that consists of the execution times for different kernels (for different data sizes) on different processors. Table 3. shown below is an example of such a lookup table.

Table 3. Lookup table example.

Kernel	Data Size	CPU (milli sec.)	GPU (milli sec.)	FPGA (milli sec.)
Matrix mult.	16000000	1967.286	0.061	76293.945
Cholesky Deco.	16000000	60.806	90.581	5.407
Matrix Inverse	698896	148.387	22.352	110.597
Matrix mult.	64000000	15487.652	0.147	610351.562
Cholesky Deco.	250000	17.064	2.749	0.093

The complete lookup table is shown in Appendix A. In this table, each row indicates the execution times of a kernel for a data size on different processors, in this case, the processors are CPU, GPU and FPGA. For example, the third row indicates the execution times in milliseconds for the matrix inversion kernel for a matrix that has 836 rows and 836 columns, therefore the data size is 836*836 i.e. 698896.

In an ideal case, if p_{min} is available (it is not executing any other kernel) then v_i is assigned to p_{min}. But p_{min} can be busy executing some other task v_k and now the policy has to make the critical decision to *wait* for this processor to be available or to assign the task to ***alternative processor*** (second best processor). APT uses ***threshold*** i.e. ***a threshold constant*** (customizable as per policies demand), that decides if v_i should be allocated to ***alternative processor*** (p_{alt}) or should the policy wait for p_{min} to be available for execution. Trying to allocate the task to an alternative processor is the difference in APT when compared to MET and this alternative processor can keep the system busy, while also reducing the waiting time.

The concept of finding the ***alternative processor*** is very important in understanding APT's functioning. If $\boldsymbol{x}$ is the execution time of kernel v_i on p_{min} (which can be found from the lookup table), then the ***threshold*** for kernel v_i, can be defined by (8)

$$threshold = \alpha * x, \quad where\ \alpha \geq 1 \tag{8}$$

where α is the customizable variable that can take any value greater than or equal to 1. Also, if v_i is dependent on some previous task (v_{prev} executing on processor p_{prev}) for data, then the transfer time for that data from p_{prev} to the contending processors is also very crucial in choosing p_{alt}. Using this, we can define the ***alternative processor*** $\boldsymbol{p_{alt}}$ as "*a*

processor for which the addition of execution and the data transfer times is less than or equal to the policy's established threshold, and is available to execute kernel v_i". The purpose of defining a threshold is to address the trade-off between waiting for the best processor and assigning the task at hand to an alternative processor. α's value determines how large or small the threshold is, which governs the degree of flexibility of the heuristic in choosing the alternative processor. As we will see, this degree of flexibility will affect the efficiency of the scheduling policy depending highly on the degree of heterogeneity of the system. The proposed algorithm for APT is formalized in Algorithm 1.

Algorithm 1. The APT Algorithm

```
1: const threshold = αx
2: while(true)do
3:      collect DFGs of all incoming jobs
4:      for (all available kernels in DFG) do
5:          p_min ← findBestProc(kernel)
6:          if there is a p_min do
7:              allocate current kernel to p_min
8:              remove current kernel from available list
9:          else do
10:             p_alt ← find2ndBestProc(kernel, threshold)
11:             if there is a p_alt do
12:                 allocate current kernel to p_alt
13:                 remove current kernel from available list
14:             end if
15:         end if
16:     end for
17: end while
```

The algorithm starts by setting the threshold for the policy and then it is always waiting for new tasks to be allocated to the processors. In line 4 we see that the policy iterates through all available kernels that are waiting for execution. The policy tries to find p_{min} for the kernel using the function *findBestProc* in line 5 and allocates the kernel to p_{min} if p_{min} is available in lines 6 and 7. Later in line 8, the kernel is removed from the list of available kernels. If p_{min} was not available, we see that the policy tries to find the alternative processor (p_{alt}) with the function *find2ndBestProc* in line 10. If it finds p_{alt}, the policy allocates the kernel to p_{alt} in line 12. And in line 13, the kernel is removed from the list of available kernels.

A closer look at the algorithm can help us understand that a larger threshold means that in the case when p_{min} is not available, the policy is willing to sacrifice on the least execution time rather than waiting. And a smaller threshold signifies that the policy is very stringent in choosing the alternative processor and is not designed to allow a lot of slack in terms of execution time of the kernel. But the overall effect of the threshold is influenced also by the heterogeneity of the system. To get the best results, a good balance of α value is to be found with respect to the heterogeneity of the system. This effect is explained in detail in chapter 4 with the experimental results. In Table 4 shown below, we see the comparison of key factors of APT with other scheduling policies.

Table 4. Summary of key properties of the scheduling policies HEFT, PEFT, SS, AG, SPN, MET and APT

	HEFT	PEFT	SS	AG	SPN	MET	APT
Scheduling policy Type	Static	Static	Dynamic	Dynamic	Dynamic	Dynamic	Dynamic
Heuristic/statistical rank calculation of kernels	Yes	Yes	Yes	No	No	No	No
Considers heterogeneity in execution times	Yes	Yes	Yes	Indirectly	No	Yes	Yes
Considers data transfer time	Yes	Yes	No	Yes	No	No	Yes
Never waits (if kernel and processor available)	Yes	Yes	Yes	No	Yes	No	No

3.2. Methodology

In this section we establish the method of evaluating the performance of scheduling policies for a heterogeneous system. With the help of the work by Skalicky *et al*.[5], we have developed a software to simulate the distributed hardware heterogeneous system, the incoming stream of applications as a work load for the system and the different scheduling policies.

The simulated heterogeneous system comprises of commercial-off-the-shelf (COTS) CPUs, GPUs and FPGAs and each communication link is based on PCI Express (PCIe). The number of processors of any type are customizable in the software and so is the communication bandwidth between the processors. This helps in creating a simulator

for any kind of heterogeneous system with different kinds of processors. For our work, we have used the system with one CPU, one GPU and one FPGA.

A stream of applications serves as an input to the scheduler of the heterogeneous system. This stream of applications can be represented as a DFG (Data Flow Graph) of kernels. An input stream can have multiple applications and each application can have multiple similar or distinct kernels. The kernels within the application can be independent of one another too and applications may have data or computational dependencies among them. This input stream can have as many applications, and there is no specific number of instances or order in which the applications occur. We have used two types of input streams for our work, 1. input stream without any dependencies and 2. input streams with dependencies, which we will henceforth refer to as DFG Type-1 and DFG Type-2 respectively. To generate each type of input stream, we have written a software which accepts for an input, a series of kernels and each kernel has its own data size. This series of kernels is then fit into the model/type of DFG, either DFG Type-1 or DFG Type-2, as needed. The series of kernels given as input, has different number of kernels and different data sizes for each kernel.

If there are a total of *n* kernels in the input, then the graph generated of DFG Type-1 will have *n-1* kernels available for execution in parallel with no data or computational dependencies (referred to as *level-1*) and only after these kernels are executed, the last n^{th} kernel is available for execution. An example of DFG Type-1 with 9 kernels can be seen in Figure 3, where all the kernels are available for execution in parallel at the same time. If

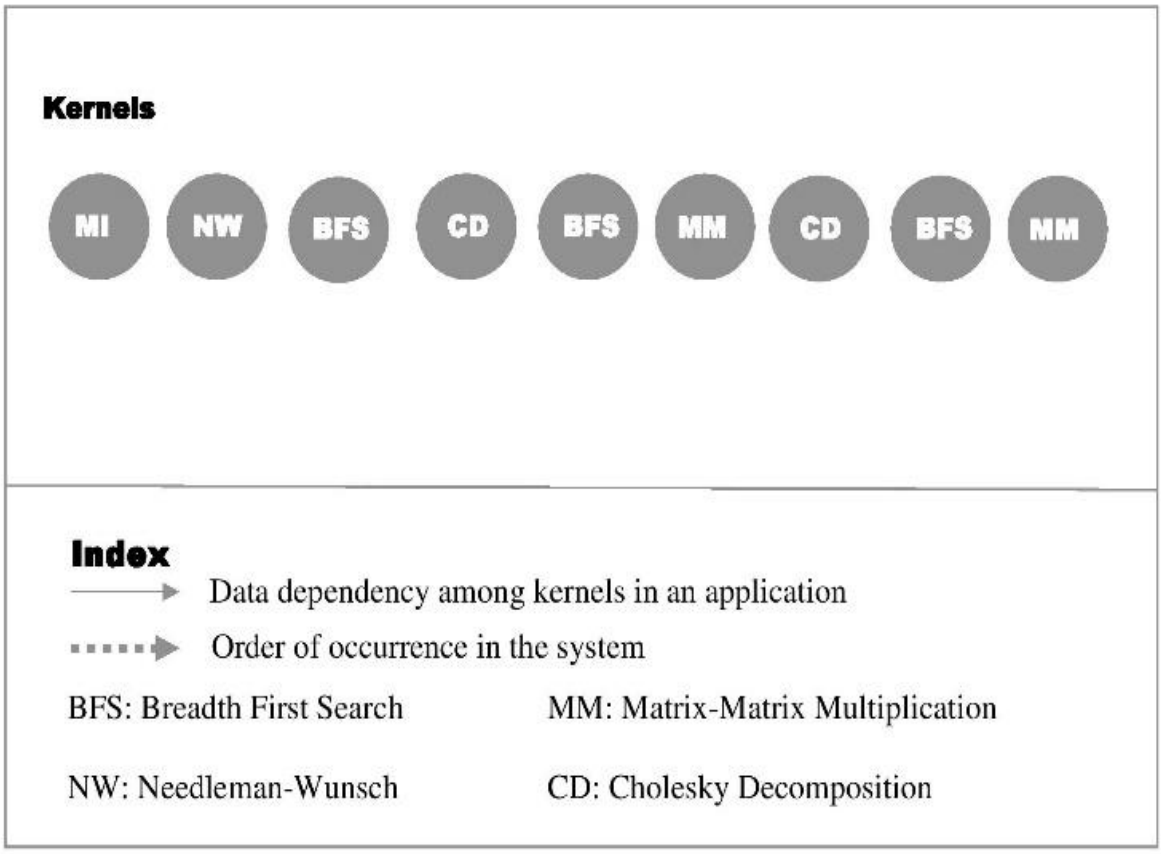

Figure 3. An example for DFG Type-1.

there are *n* kernels in the input, then the graph generated of DFG Type-2 will have data and computational dependencies among kernels as shown in figure C. There are individual kernels and a group of kernels with computational and data dependencies. There is also a *kernel graph block* with a diamond like structure with one kernel each at the top and bottom, and multiple independent kernels in the middle. There are three such *kernel graph blocks* in any graph in Figure 3, where 9 kernels are available for execution in parallel and after the execution of these 9 kernels, the 10^{th} kernel is available for execution. If there are

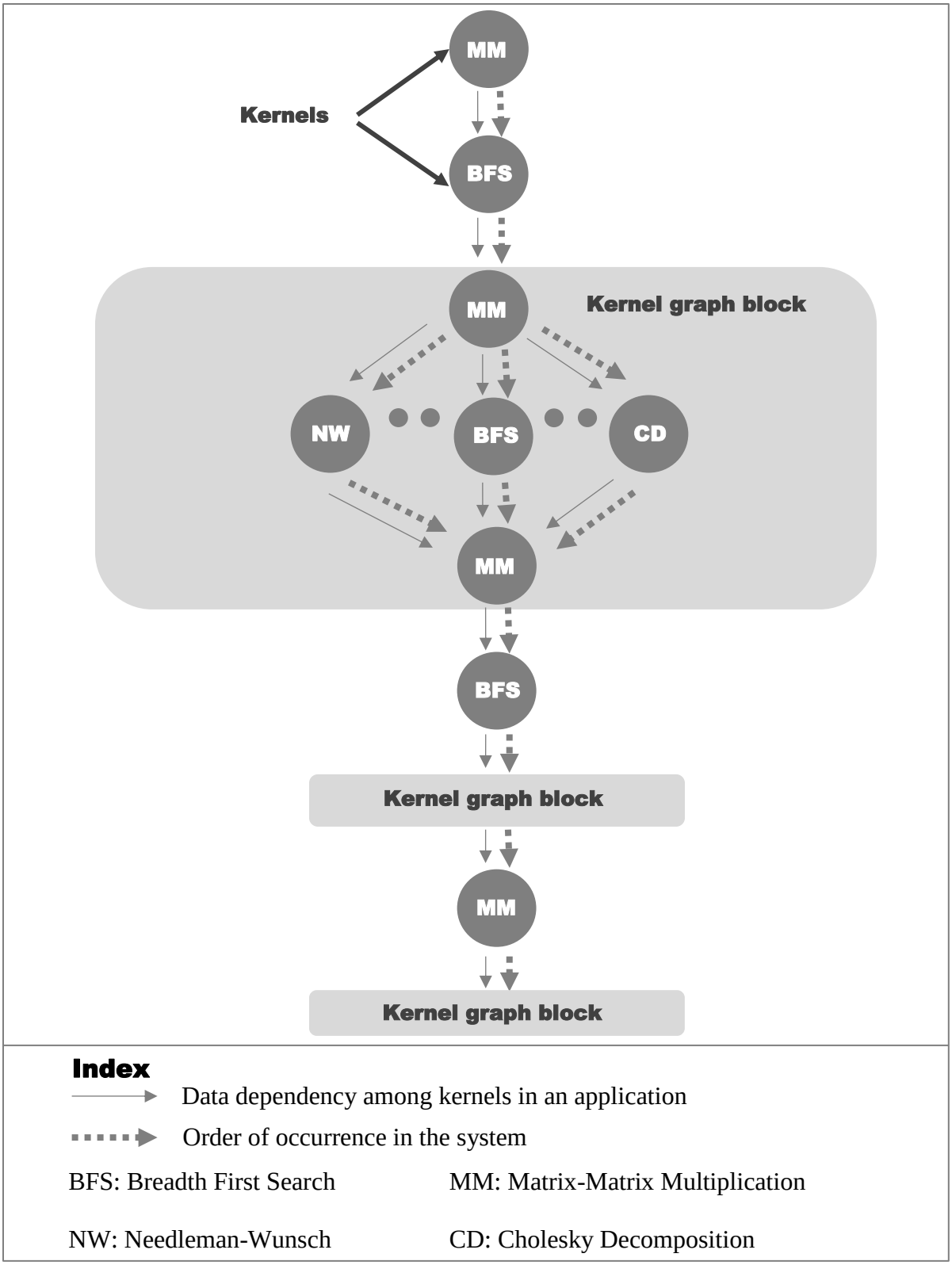

Figure 4. An example for DFG Type-2.

a total of *n* kernels in the input, then the graph generated of DFG Type-2 will have data and computational dependencies among kernels as shown in figure C. There are individual kernels and a group of kernels with computational and data dependencies. There is also a *kernel graph block* with a diamond like structure with one kernel each at the top and bottom, and multiple independent kernels in the middle. There are three such *kernel graph blocks* in any graph of DFG Type-2, just as shown in figure 4. When the number of kernels in the input changes, the structure remains the same, for both the types of graphs. The only thing that changes is the number of kernels in *level-1* in DFG Type-1 and the independent kernels in *kernel graph blocks* of DFG Type-2.

The graphs generated using the above-mentioned software serve as the input to the scheduling policy. These graphs have different kernels in different orders and each kernel has different data size, therefore ensuring that the scheduling policies are evaluated without

Table 5. Kernels chosen in our work

Work by	Kernels	Dwarf
Krommydas *et al.*[am]	Needleman Wunsch	Dynamic Programming
	Breadth First Search	Graph Traversal
	Speckle Reducing Anisotropic Diffusion	Structured Grids
	Gaussian Electrostatic Model	N-Body
Skalicky *et al.*[u]	Cholesky Decomposition	Dense and Spare Linear Algebra
	Matrix-Matrix Multiplication	
	Matrix Inversion	

any bias and the results can be extrapolated to any stream of applications. We have 10 input graphs for both, DFG Type-1 and DFG Type-2; generated using the software described above and each graph of a type has different order and number of kernels. The kernels that are chosen to be a part of the workload for the stream of applications are shown in Table 5.

We give a brief understanding of these kernels in the following paragraphs.

- ***Needleman-Wunsch (NW)*** - The Needleman-Wunsch algorithm is a dynamic programming algorithm for optimal sequence alignment [38]. This algorithm is a nonlinear global optimization method that is used for amino acid sequence alignment in proteins. Because the Needleman-Wunsch algorithm finds the optimal alignment of the entire sequence of both proteins, it is a global alignment technique, and cannot be used to find local regions of high similarity.
- ***Breadth First Search (BFS)*** - Breadth First Search is an algorithm to traverse a graph in search of a node or a path, usually starting from its root node. In this algorithm, all immediate unvisited neighbors are inspected. Subsequently, for each of these neighbors, their own unvisited immediate neighbors are visited, eventually traversing the entire graph. The traversing is terminated depending on the problem statement, for example, if the algorithm is used to find for a particular node, then the algorithm terminates if the node is found or if the entire graph is traversed and still the node wasn't found.
- ***Speckle Reducing Anisotropic Diffusion (SRAD)*** - Speckle Reducing Anisotropic Diffusion, also known as SRAD is an algorithm based on partial differential equations and used to remove speckles from images and is widely used in ultrasonic

and radar imaging applications. It is the edge-sensitive diffusion for speckled images. This is similar in ways that the conventional anisotropic diffusion is the edge-sensitive diffusion for images corrupted with additive noise. Apart from perfectly preserving edges, SRAD also enhances edges by inhibiting diffusion across edges and allowing diffusion on either side of the edge.

- ***Gaussian Electrostatic Model (GEM)*** - Electrostatic interactions are a very important factor in determining properties of biomolecules. The ability to compute electrostatic potential generated by a molecule is often essential in understanding the mechanism behind its biological function such as catalytic activity or ligand binding. Gaussian Electrostatic Model, also referred to as GEM, calculates the electrostatic potential of a biomolecule as the sum of charges contributed by all atoms in the biomolecule owing to their interaction with a surface vertex (two sets of bodies)
- ***Cholesky Decomposition (CD)*** - The Cholesky decomposition [39] of a positive definite matrix A is an upper triangle matrix U with only positive diagonal values such that it satisfies (9).

$$A = U^T U \tag{9}$$

- ***Matrix - Matrix Multiplication (MatMul)*** **–** The purpose of the kernel is singular i.e. multiplying two matrices, and the operations (instructions) involved in this kernel are very simple too. but, it is one of the most highly used kernels in a variety of domains including image processing, machine learning, computer vision, Finite State Machines, and many more.

- ***Matrix Inverse (MI)*** – Like *MatMul*, this again is one of the most widely used kernels across domains. The inverse of the matrix A is denoted by A^{-1} such that,

$$A\,A^{-1} = I \tag{10}$$

where *I* is an identity matrix of the same dimensions as that of A.

We use these kernels in the stream of applications and their execution times in the lookup table. These execution times used for our work for the following hardware platform specifications:

Table 6. Hardware platform specifications.

Work by	CPU	GPU	FPGA
Krommydas *et al*.[24]	AMD Opteron 6272 16 cores @2.1GHz	AMD Radeon HD 6550D @ 600MHz	Xilinx Virtex-6 LX760
Skalicky *et al*. [5]	Intel Core i7 2600 3.4GHz 16GB DDR3 @1.333Gbps	Nvidia Tesla K20 706MHz 5GB GDDR5 @5.2GHz	Xilinx Virtex 7 VX485T, VC707 1GB DDR3 @ 1600Mbps

In our work, we have made a generalization that *the execution time for any given kernel belongs to the category of the platform*. What this means is that, we have the execution times on the Intel Core i7 2600 CPU for the kernel *matrix-matrix multiplication* from the work of Skalicky *et al*.[5], and not on AMD Opteron 6272 (CPU used by Krommydas *et al*.[24]); but we will assume that this is the execution time for the category CPU, irrespective of the exact CPU configuration. Similarly, we will also assign the execution times to the categories GPU and FPGA, and not the specific configuration of hardware.

Using PCIe 2.0 the data rate per lane is 500MBps, we varied the number of lanes to be 8 and 16 so that we can understand the effect of varying the data transfer rates on the performance of APT. With 8 lanes (x8) this would achieve an approximate throughput of 4GBps (500×8) and with 16 lanes (x16) this would achieve an approximate throughput of 8GBps (500×16). In our work, we maintain the data transfer rates between all processors to be the same i.e. if the rate is 4GBps, then it is the same from the CPU to GPU, GPU to FPGA and CPU to FPGA.

Once the system starts receiving computations that are to be executed, it is the job of the scheduler to assign tasks to a processor and this decision of assignment of any task to a particular processor is made by the scheduling policy. Each scheduling policy has its own strategy to make assignments. This scheduler also has access to a lookup table which has real execution times of a variety of kernels (each belonging to some dwarf category) from the works of Skalicky *et al*.[5] and Krommydas *et al*.[24] for multiple data sizes on the different processors. Using these execution times is one key difference in our work when compared to other efforts. *This lookup table is a medium of generalizing and estimating an approximate execution time of any kernel on any kind of processor in the heterogeneous system*. Following its own strategy and using this lookup table, in the case of dynamic policies, the scheduler assigns all the incoming tasks and at time T, finally generates a log of the schedule in which the tasks were assigned to different processors. But for static policies, the scheduler generates a log of the schedule that it had generated beforehand over multiple iterations of constraint optimization.

Other than creating a schedule for a given stream of applications, the simulator also calculates a few statistical metrics like,

1. ***total execution time (makespan)***- total time the system was busy executing the said stream of applications,
2. ***compute time per processor*** - time for which each processor was busy,
3. ***transfer time per processor*** - time for which each processor was engaged in transferring data,
4. ***idle time*** - time for which each processor was idle,
5. ***number of occurrences of better solutions*** – number of times the policy performed better than other policies,
6. ***total λ delay*** – comprises of three factors (a) the scheduling delay to process which task should be assigned to which processor, (b) communication delay from the scheduler to the processor to begin processing and provide the necessary information, (c) dependencies on kernels that are being executed in another processor, but have not completed yet.
7. ***average λ delay*** - It can be calculated as follows:

$$\lambda_{avg} = \frac{\lambda_{total}}{N} \tag{11}$$

   where N is the number of times a delay occurred.

8. ***standard deviation of λ delay*** - It can be calculated with the formula:

$$\lambda_{stddev} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\lambda_i - \lambda_{avg}\right)^2} \tag{12}$$

   where N is the number of times there was a delay and λ_i is the delay for each delay occurrence.

Chapter 4 Experimental Results

4.1. Comparison of schedule generated by APT and MET

To start with, we show an example of a simple workload of DFG Type-1 with the α value equal to 8, that illustrates how the schedule for APT differs when compared with MET. Also, to simplify the example, we do not consider transfer times. In this example, there are only 3 types of kernels ***bfs*** (3 occurrences)***, nw*** (1 occurrence) and ***cd*** (1 occurrence). The actual execution times of these kernels on different processors are as shown in table 7. It is clear that these kernels have far apart execution times on different processors, making them key contenders for our work and they also make it easy to explain the example.

Table 7. Execution time of different kernels.

Kernel	CPU execution time (ms)	GPU execution time (ms)	FPGA execution time (ms)
NW	1.12×10^{2}	1.46×10^{2}	3.97×10^{2}
BFS	3.32×10^{2}	1.73×10^{2}	1.06×10^{2}
CD	17064×10^{-4}	2749×10^{-3}	93×10^{-3}

The first column indicates the kernels allocated to CPU, the second column to GPU and the third column to FPGA. The last column is the current time stamps when there is a new allocation to a processor or when a kernel ends execution on any processor. Therefore, each row represents a ***state*** that the system is currently in. The value in last column for each row is when the system entered that state of kernel allocation. The states with the gray background are the ones where APT makes a different decision when compared with MET.

MET Schedule

```
CPU:0-nw            GPU: idle      FPGA:1-bfs          0.0
CPU:0-nw            GPU: idle      FPGA:2-bfs        106.0
CPU: idle           GPU: idle      FPGA:2-bfs        112.0
CPU: idle           GPU: idle      FPGA:3-bfs        212.0
CPU: idle           GPU: idle      FPGA:4-cd         318.0

End time: 318.093
```

APT Schedule ($\alpha = 8$)

```
CPU:0-nw       GPU:2-bfs           FPGA:1-bfs          0.0
CPU:0-nw       GPU:2-bfs           FPGA:3-bfs        106.0
CPU: idle      GPU:2-bfs           FPGA:3-bfs        112.0
CPU: idle      GPU: idle           FPGA:3-bfs        173.0
CPU: idle      GPU: idle           FPGA:4-cd         212.0

End Time: 212.093
```

Figure 5. MET and APT schedule example.

The kernel ***bfs*** has the FPGA processor as p_{min} but it is busy executing kernel number 1 (***bfs***) on FPGA. Now there is a ***bfs*** kernel (kernel number 2) that needs to be scheduled so APT looks for p_{alt} and the GPU satisfies the condition of threshold ($173 < 8 \times 112$) i.e. execution time on GPU is less than the threshold ($\alpha \times$ execution time on FPGA). Therefore, APT assigns kernel number 2 (***bfs***) to GPU. A detailed analysis for APT's choices in different experiments in our study is described in Appendix B.

4.2. Performance comparison of total execution times.

4.2.1 Input stream: DFG Type-1

In this section we discuss the total execution times generated by all 7 scheduling policies for 10 graphs of DFG Type-1 with 4 GBps data transfer rates between all processors. In Table 8, we see the total execution times for all polices and the α value in APT is set to 1.5. Each row in the table indicates the total execution time for that graph. The cells colored yellow are the ones with the least execution times and the red cells are

Table 8. Total computation time in milliseconds for DFG Type-1 by all policies (α=1.5 for APT).

Graph	APT	MET	SPN	SS	AG	HEFT	PEFT
1	8298	8006	9330	19492	37952	9895	10164
2	27684	27684	42668	43383	591817	29319	27872
3	18991	18991	95411	98063	593087	20880	20197
4	53742	53742	98174	98604	113883	56151	54171
5	49425	49425	53283	52788	587004	50748	50362
6	96956	96956	638213	630378	588153	98507	96956
7	69549	69549	70591	628454	586129	72193	70478
8	90130	90130	631282	632797	607355	92920	90443
9	129578	129578	638824	642936	698367	131629	130148
10	166430	166430	641107	642246	190435	169177	167152

the highest execution times. We see that APT performs better than all other scheduling policies for 9 out of 10 graphs and the execution times are exactly the same as that of MET for these graphs. APT does not perform as much as MET for the first graph. This is because the randomly generated graph happened to have a lot more kernels with relatively smaller execution times, like matrix multiplication, matrix inverse, Cholesky decomposition; and

lesser kernels with larger execution times, like NW, BFS, SRAD and GEM. The generated schedule for this graph shows that at one instance, the order of kernels causes the policy to assign a kernel with higher execution to execute on an alternative processor, thus an increased total execution time. Knowing that the chosen kernels in our study have a lot of heterogeneity and so do the processors, the results match with the theoretical expectation of having similar behavior when compared to MET. We now understand that an α value of

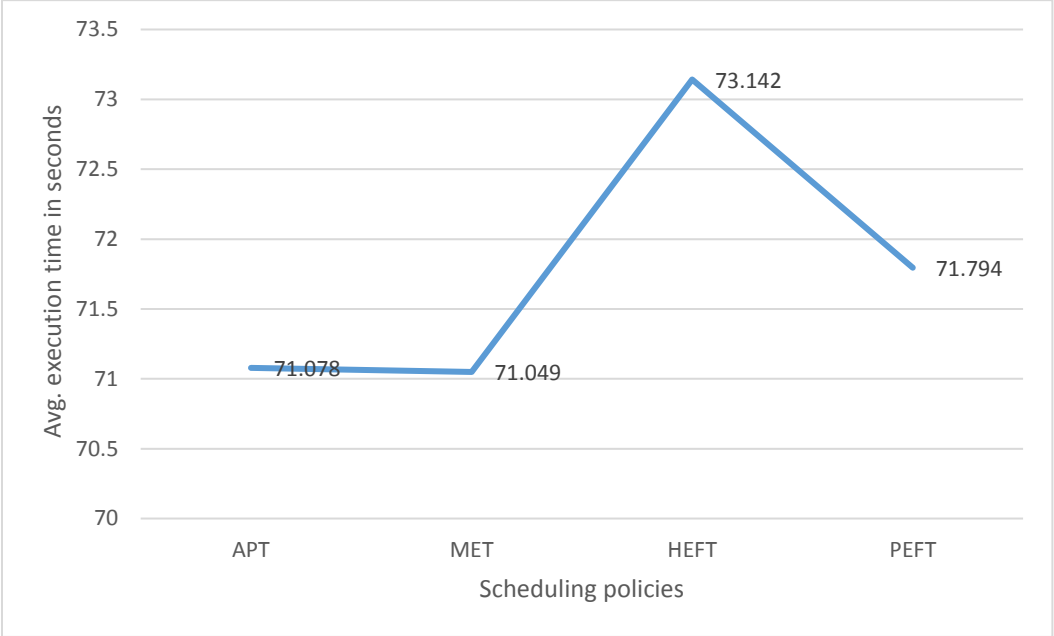

Figure 6. Avg. execution time in seconds for top 4 policies of DFG Type-1.

1.5 is too small to offer the APT policy any kind of flexibility in the case when p_{min} for any kernel v_i is busy. Also, really close execution times for a kernel v_i on very different processing platform, defeats the purpose of choosing heterogeneous systems. Figure 6 shown above has the average execution time of all the 10 graphs for the top 4 performing scheduling policies when APT has α set to 1.5. This figure shows a comparison of the

performance of APT with the other top performing policies. We can see that the best performing dynamic policies are APT and MET; and following them are the static policies HEFT and PEFT. Bear in mind that HEFT and PEFT are static scheduling heuristics. However, being dynamic, MET and APT are performing better than the static ones. On the other hand, as noted above, we do not attain any improvement by setting up this threshold in our new proposed policy. The reason for this is that the α value in this case is too small to make a difference, given the degree of heterogeneity of our system and applications. With the little extra margin of 0.5x the best-case time, even when there is an assignment to a p_{alt}, the impact on the overall execution time is minimal. For that reason, we explore different values for α. The goal is to keep α at a level in which the assignment to an alternative processor will reduce the wait of the workload without hurting the performance of that kernel by too much. In other words, an α value that is too small limits the cases in which an alternative processor will be chose, while an α value that is too high will constantly assign to significantly slower processors, hurting overall performance. For that reason, the degree of heterogeneity and α values go hand-in-hand to provide best performance.

We understand that an increase in α is a mechanism of allowing more kernels to have the chance to be allocated to an alternative processor p_{alt} when p_{min} is busy. Considering this, we setup a hypothesis, in which we state that *"If we increase the α value, the makespan also decreases to a point, after which, the makespan keeps increasing with an increase in α value"*. In defense of the hypothesis, the decrease in makespan can be explained by the flexibility (to choose p_{alt}) that the policy attains with the increase in α value. And the increase in makespan can be attributed to the fact that after the point of

inflection ($threshold_{brk}$), the difference between the execution time of kernels on p_{alt} and p_{min} is large enough to hurt the performance than benefiting it. We believe in this hypothesis, because this decrease in makespan will signify that the strategy to choose an alternative processor (p_{alt}) is better than waiting for the best processor (p_{min}).

To prove this hypothesis, we vary the α values to be 1.5, 2, 4, 8 and 16; and compare the performance of APT in Figure 7. We have a bar for each α value, each bar representing the average performance for that α value. But as expected from theory, average execution

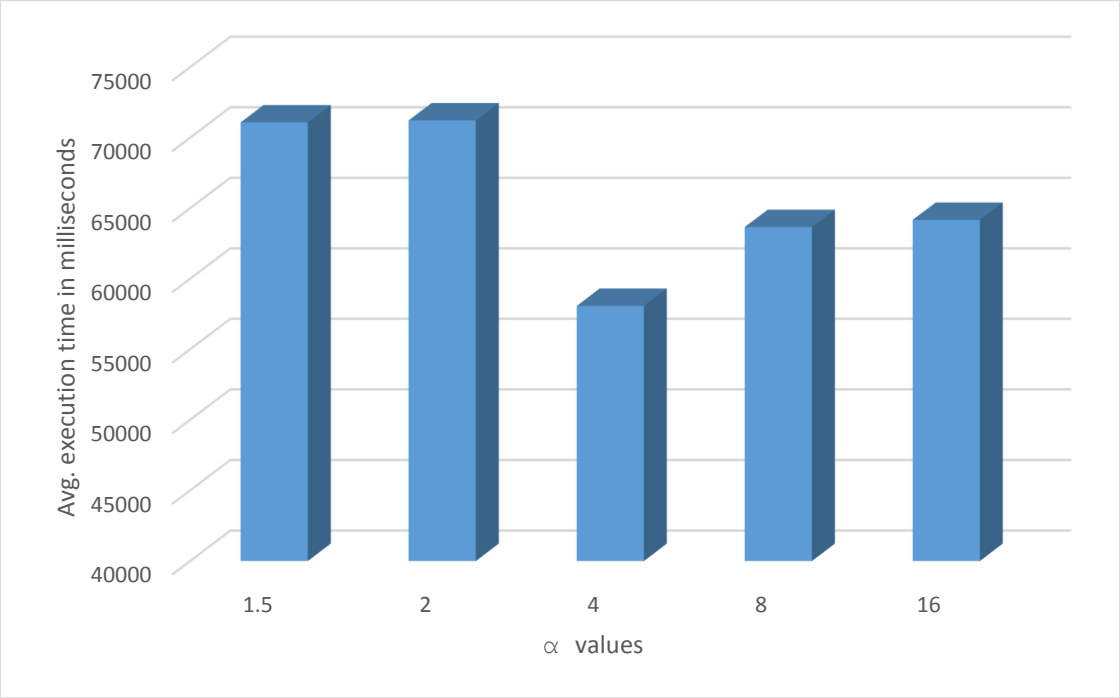

Figure 7. Avg. performance of APT for DFG Type-1 on varying α and transfer rate .

time decreases till α reaches a point (in this case 4), after which the average execution time increases, we refer to this trend as the *valley*. α = 4 can be referred to as $threshold_{brk}$, APT gives the least average execution time and it outperforms all other scheduling policies for 9 out of 10 graphs. In Figure 8, we show the difference in the execution time for the

different experiments of DFG Type-1 for the α value of 4. The average execution time falls drastically when compared to the closest performing dynamic policy, MET; in 9 out of 10 graphs. The average execution time falls 16% in average when compared to the closest performing dynamic policy, MET. This means that a ***change in the α value changes the order in which the kernels are executed and the assignments are made to different processors than the one with the least execution time***.

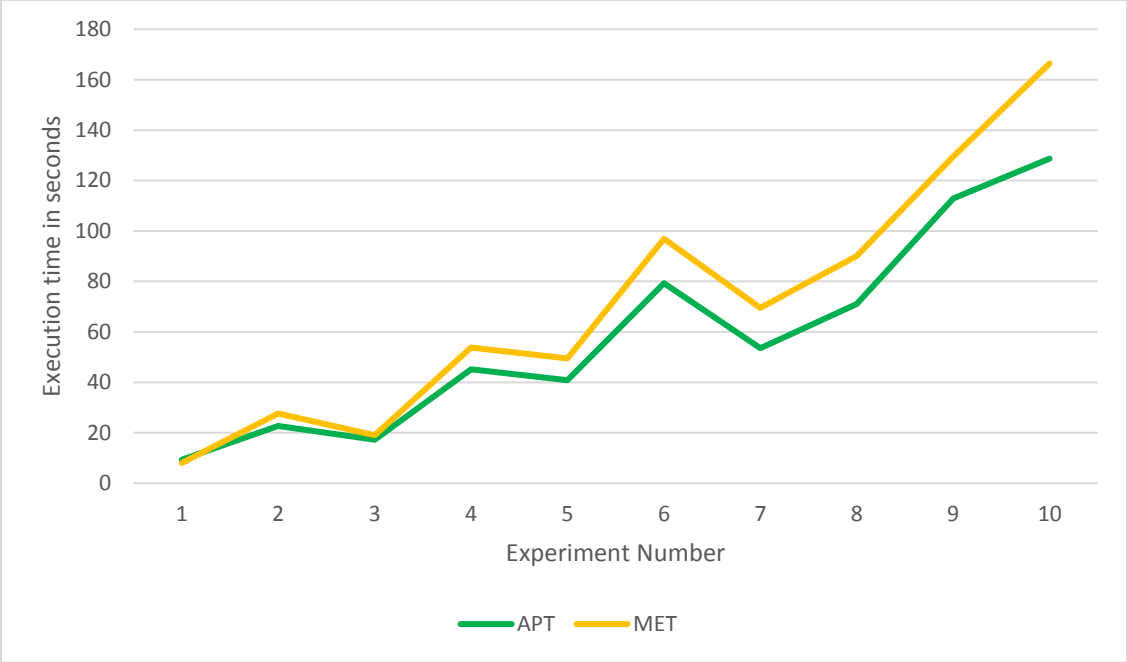

Figure 8. Execution time of experiments of DFG Type-1 for MET and APT (α=4)

4.2.2 Input stream: DFG Type-2

In Table 9, we see the total execution times generated by all 7 scheduling policies (with α = 1.5 for APT) for 10 graphs of DFG Type-2 with 4 GBps transfer rates between

Table 9. Total computation time in milliseconds for DFG Type-2 by all policies (α=1.5 for APT).

Graph	APT	MET	SPN	SS	AG	HEFT	PEFT
1	10274	10274	10309	30518	788846	10702	10608
2	30617	30617	229508	196084	238349	31980	30866
3	20637	20637	125312	37751	933473	21521	21226
4	56769	56769	286415	289913	1791863	58806	57721
5	52674	52674	1006508	229536	809491	53613	53442
6	97807	97807	798962	808662	999713	99480	98253
7	72686	72686	232781	212426	645713	74758	73421
8	93928	93928	250180	757902	906603	96817	94493
9	131875	131875	753455	252985	1871693	133688	132332
10	172185	172185	757131	836395	560595	174562	172959

all processors. Each row in the table indicates the total execution time for that graph. We see that APT outperforms all other scheduling policies for all the graphs and the execution times are the same as that of MET.

Figure 8 shown below has the average execution time of all the 10 graphs for the top 4 performing scheduling policies when APT has α set to 1.5. The performance order of policies from the quickest to the slowest is MET, APT, PEFT and HEFT. Of these top 4 policies, HEFT and PEFT are static.

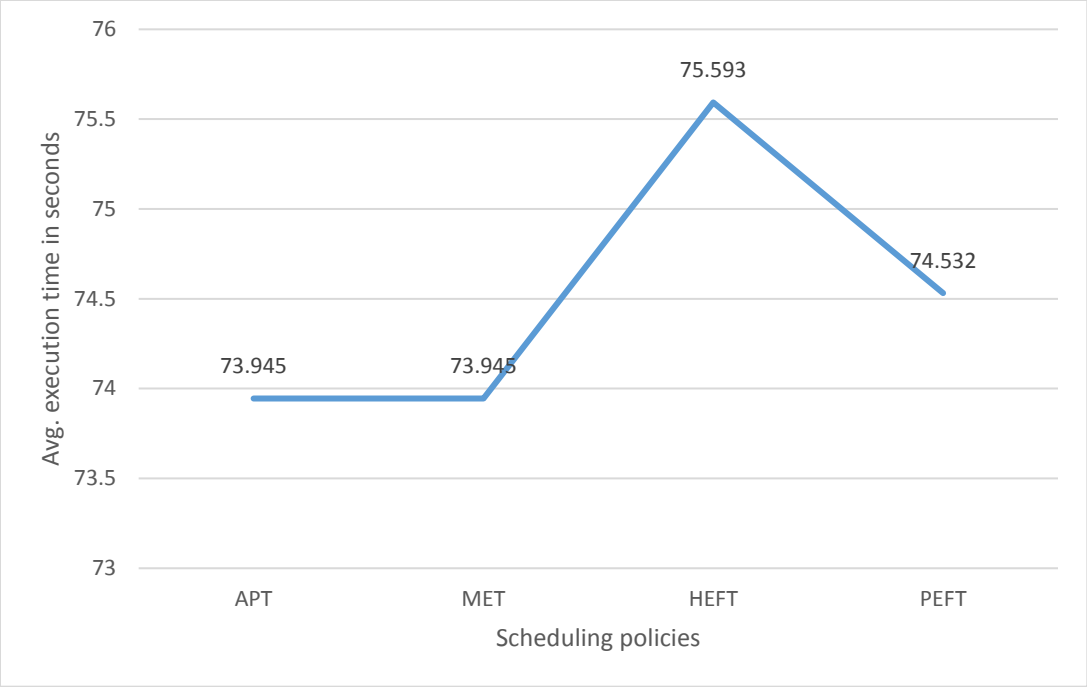

Figure 8. Avg. execution time in seconds for top 4 policies of DFG Type-2.

With α having a value as small as 1.5, we see that APT and MET have exactly the same values for total execution times. Such mimicking behavior is expected from our policy APT because a smaller α indicates that if the kernels have really large heterogeneities, the policy will always look for the best processor p_{min}. But as we increase the α value to 4, as in the case of table 10, we see that the behavior of APT starts changing from that of MET. Also now we see that for ***9 out of 10 graphs have better results with APT when compared to other policies***.

Table 10. Total computation time in milliseconds for DFG Type-2 by all policies (α=4 for APT).

Graph	APT	MET	SPN	SS	AG	HEFT	PEFT
1	10090	10274	10309	30518	788846	10702	10608
2	26554	30617	229508	196084	238349	31980	30866
3	20683	20637	125312	37751	933473	21521	21226
4	50443	56769	286415	289913	1791863	58806	57721
5	41940	52674	1006508	229536	809491	53613	53442
6	82955	97807	798962	808662	999713	99480	98253
7	58631	72686	232781	212426	645713	74758	73421
8	78124	93928	250180	757902	906603	96817	94493
9	115916	131875	753455	252985	1871693	133688	132332
10	137491	172185	757131	836395	560595	174562	172959

We also vary the α values and compare the performance of APT in Figure 9. We have two bars for each α value, each bar representing the average performance for a different data transfer rate. There is a little difference in the average execution time with an increase in the data transfer rate. This can be attributed to the fact, that when the transfer rate increases, the transfer time decreases and therefore the probability that the processor can be p_{alt} increases. Because of this, we see the *valley* for data transfer rate of 4 GBps and 8 GBps with $threshold_{brk}$ at $\alpha = 4$ and APT outperforms all other policies for 9 out of 10 graphs. This means that it is not possible to have APT always outperform other policies for one given α value. The performance is also dependent on the heterogeneity of the workload and the transfer rates between processors. Also in Figure 10, we see the difference in execution times for different experiments of DFG Type-2 for the α value of 4.

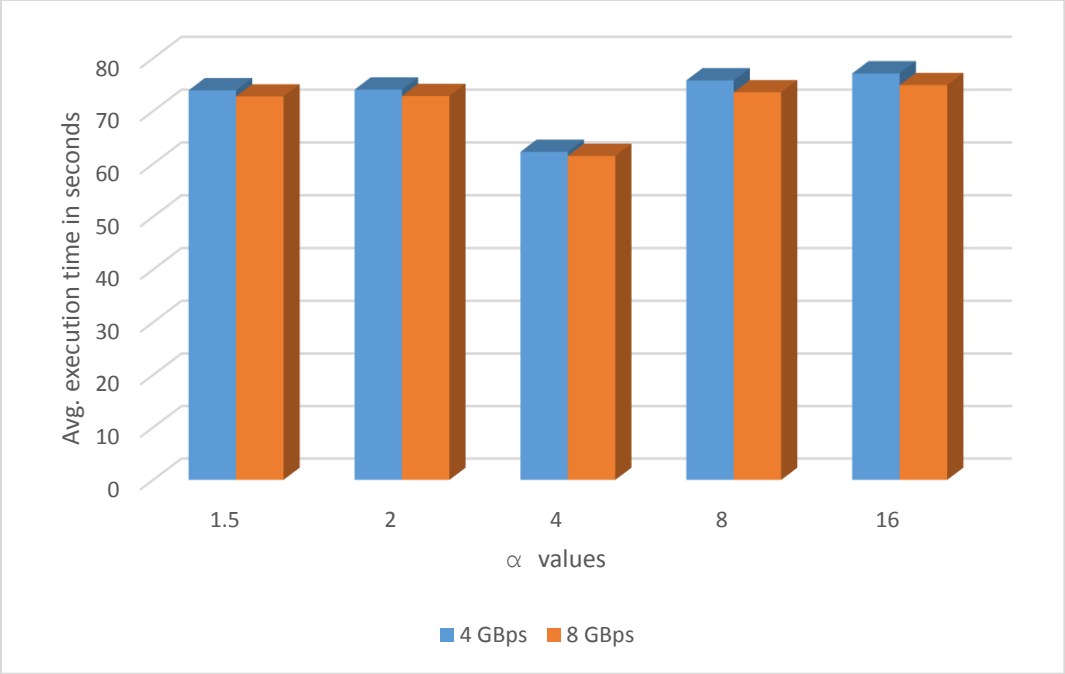

Figure 9. Avg. performance of APT for DFG Type-2 on varying α and transfer rate

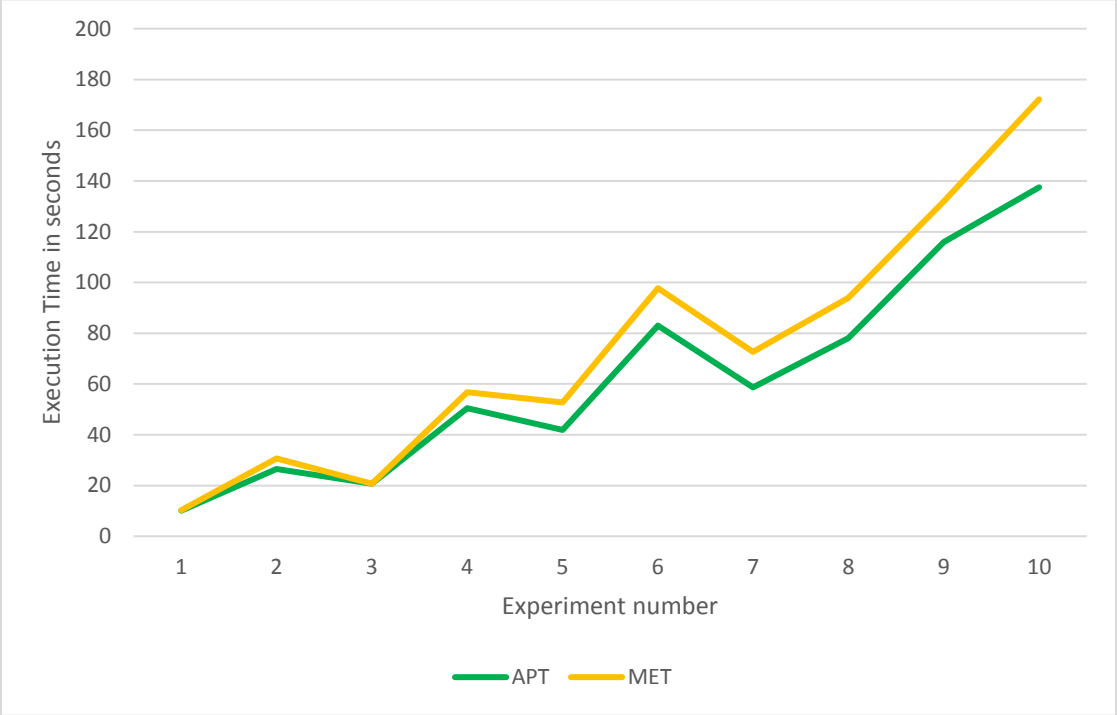

Figure 10. Execution time of experiments of DFG Type-2 for MET and APT (α= 4)

4.3. Performance comparison of λ delay times.

Understanding this, we dwell further into comparing the performance of scheduling policies and ***total scheduling delay*** is a very important metric for evaluating the performance of scheduling policies.

4.3.1 Input stream: DFG Type-1

In this section we discuss the λ delay times generated by all 7 scheduling policies for 10 graphs of DFG Type-1 with 4 GBps data transfer rates between all processors. In table 11, we see the λ delay times for all polices and the α value in APT is set to 4. Each row in the table indicates the total λ delay time for that graph. We see that APT performs

Table 11. Total λ delay in milliseconds for DFG Type-1 by all policies (α=4 for APT).

Graph	APT	MET	SPN	SS	AG	HEFT	PEFT
1	4907	3877	15852	11486	15102	7154	10586
2	19092	25690	118454	37532	585761	28025	43828
3	14260	17035	266316	81902	579353	19733	34051
4	40257	49629	278508	89813	87195	54119	95552
5	40600	49289	138741	35640	460003	51487	64707
6	78130	96722	1903781	608785	585600	98958	127305
7	52583	69381	207626	606860	571359	73335	110667
8	65439	85992	1873296	617699	580668	89856	147226
9	111143	129485	1911829	628342	450206	132775	203875
10	123690	164457	1902519	626353	169364	169065	250403

better than other scheduling policies in 8 out of 10 graphs. Just like total execution times, we observe the λ delay times to exhibit the *valley* shape on varying the α values. This behavior can be seen in the figure 11 shown below.

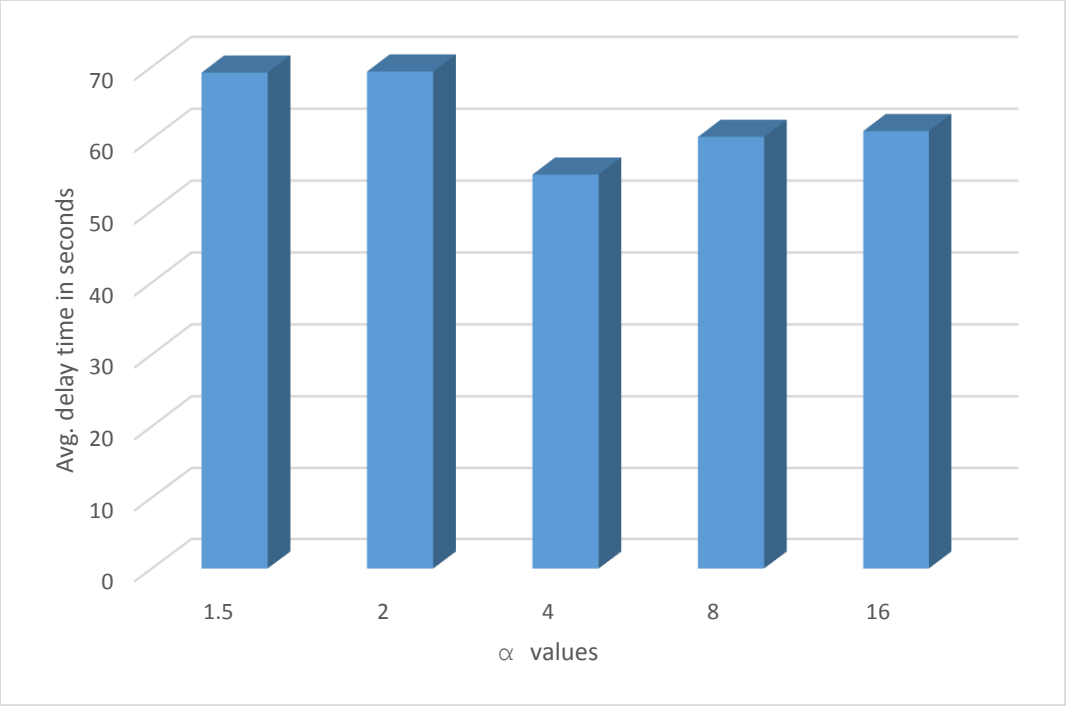

Figure 11. Avg. λ delay times in seconds of APT for DFG Type-1 on varying α and transfer rate.

4.3.2 Input stream: DFG Type-2

In table 12, we see the λ delay times generated by all 7 scheduling policies (with α = 4 for APT) for 10 graphs of DFG Type-2 with 4 GBps data transfer rates between all processors. Each row in the table indicates the λ delay time for that graph. We see that the λ delay time of APT is lesser than all other policies for all the 10 graphs. But one key observation we find here is the really huge time for the policy SPN. This means that because of the policies strategy to assign the process with the shortest processing time, the dependencies in the graphs, add a lot of waiting time. Also, the policy always assigns if there are available kernels and free processors, so the policy assigns the kernels to the worst processors and has the highest scheduling delay along with the some of the highest total

execution times. And in figure 13, we observe similar trend as in the case of average total execution times for graphs of DFG Type-2. The $threshold_{brk}$ for the transfer rates is at α = 4.

Table 12. Total λ delay in milliseconds for DFG Type-2 by all policies (α=4 for APT).

Graph	APT	MET	SPN	SS	AG	HEFT	PEFT
1	6561	7005	45667	27048	757575	6836	11185
2	19776	30716	3181750	168934	237605	31596	39616
3	34000	34162	972620	49141	1582343	34708	41819
4	37290	53122	3323206	243724	1742485	55448	90249
5	42340	54198	14373239	202793	797082	53856	66444
6	80980	102176	9607702	774055	1263065	104203	128984
7	64702	84197	3232597	175919	883066	76115	103024
8	91108	109048	3557879	751260	972178	107823	150583
9	109379	132673	4102061	225416	1364818	134360	202757
10	132489	173588	13902335	803735	514471	175891	254743

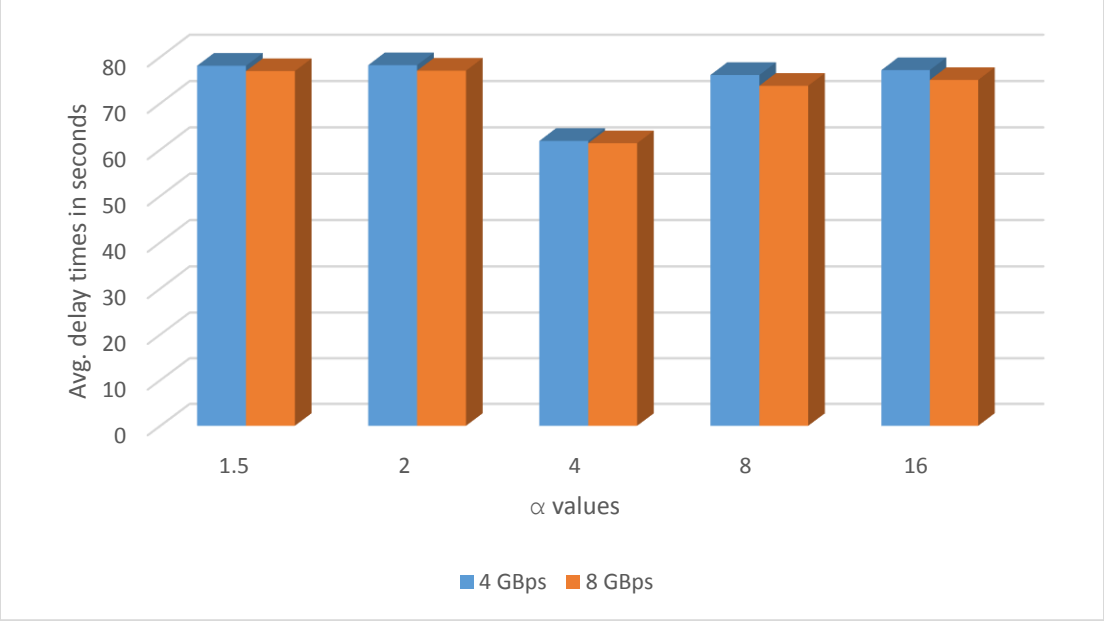

Figure 12. Avg. λ delay times in seconds of APT for DFG Type-2 on varying α and transfer rate.

4.4. Evaluation of performance enhancement.

After a thorough analysis, we list down the percentage improvement in the average computation time for all graphs (for a given α value). The improvement for total execution time and scheduling delay (λ delay) is calculated using the formula in (13) and (14) respectively. For better understanding of comparison, the second best policy can only be a dynamic policy like APT.

$$Improvement_{exec} = \frac{Avg.exec.time_{2nd\ best\ policy} - Avg.exec.time_{APT}}{Avg.exec.time_{2nd\ best\ policy}} * 100 \quad (13)$$

$$Improvement_{\lambda\ delay} = \frac{Avg.\lambda\ delay_{2nd\ best\ policy} - Avg.\lambda\ delay_{APT}}{Avg.\lambda\ delay_{2nd\ best\ policy}} * 100 \quad (14)$$

In table 13, we show the execution and λ delay improvements observed for different α values of 1.5, 2, 4, 8 and 16; for both the types of graphs, DFG Type-1 and DFG Type-2. The average execution and delay times are for the case when the transfer rate is 4 GBps. As observed from the graphs show before, the average execution times and λ delay times are not very different for the transfer rates 4 GBps and 8 GBps, therefore the difference in the improvement is also negligible. We see that for α value of 4, the average performance of APT is better for both the types of graphs. The negative values in the table indicate that the second best dynamic policy is better than APT for that α value. This is the case for α values 1.5 and 2 in the case of DFG Type-1 graphs and α values of 2, 8 and 16 in the case of DFG Type-2 graphs. This means that the performance of the policy is dependent on the heterogeneity of the hardware system and choosing the right α value is key for optimal performance.

Last, we can see that the percentage of improvement is higher for λ than for the overall execution time, as it is expected since we are making quicker assignments at the cost of lower performance for the specific kernel. The key point is that overall, the performance benefits from this lower wait period and assignment to second-best processors.

Table 13. Improvement metrics for APT with respect to different types of graphs.

	DFG Type – 1		**DFG Type – 2**	
α	*Improvement* $_{exec}$	*Improvement* $_{\lambda\ delay}$	*Improvement* $_{exec}$	*Improvement* $_{\lambda\ delay}$
1.5	-0.1	-0.044	0	0
2	-0.298	-0.256	-0.178	-0.163
4	18.223	20.455	15.771	20.778
8	10.347	12.875	-2.538	2.555
16	9.628	11.710	-4.326	1.206

Chapter 5 Conclusion

As heterogeneity tends to be present high performance computing and the diversity of platforms grows, it is important to evaluate and rethink the role that the degree of heterogeneity has in scheduling tasks across platforms. In this paper, we have presented a scheduling policy with added flexibility to assign kernels to hardware platforms. When the optimal processor is busy, the kernel next in the execution queue can be assigned to an alternative processor with higher expected execution time. How much execution time we are willing to sacrifice is pondered by a threshold that varies depending on the degree of heterogeneity of the system. This idea was tested through simulation on a CPU-GPU-FPGA system using real, measured execution and transfer times for each kernel and data size. The conclusion is that the threshold must be carefully tuned in order to attain performance improvements, but overall our Alternative Processor within Threshold approach can reduce execution time by 16% and 18% in average when compared to the second-best scheduling policy for workloads with and without data dependencies respectively. In the future, we will consider the remaining execution time in the optimal processor before deciding whether to assign to an alternative processor, as part of the scheduling heuristic, which will improve our current savings.

Chapter 6 Appendices

6.1. Appendix A

Table 14. Complete lookup table.

Kernel	Data Size	CPU	GPU	FPGA
Matrix Multiplication	250000	29.631	0.062	149.011
	698896	131.183	0.061	696.512
	1000000	220.806	0.061	1192.092
	4000000	259.291	0.062	9536.743
	16000000	1967.286	0.061	76293.945
	36000000	6676.706	0.106	257492.065
	64000000	15487.652	0.147	610351.562
Matrix Inverse	250000	42.952	9.652	24.247
	698896	148.387	22.352	110.597
	1000000	235.810	29.078	188.188
	4000000	432.330	129.156	1482.717
	16000000	40636.878	596.582	11770.520
	36000000	133917.655	1702.537	39623.932
	64000000	312902.299	3600.423	93802.080
Cholesky	250000	17.064	2.749	0.093
Decomposition	698896	86.585	4.940	0.258
	1000000	6.284	6.453	0.361
	4000000	86.585	21.219	1.382
	16000000	60.806	90.581	5.407
	36000000	132.677	220.819	12.194
	64000000	307.539	458.603	21.543
Needleman Wunsch	16777216	112	146	397
BFS	2034736	332	173	106
SRAD	134217728	5092	1600	92287
GEM	2070376	21592	4001	585760

6.2. Appendix B

The following table has the analyses of how different APT behaves for DFG Type-1 when compared to MET which is the closest counterpart. In the first column, we list the experiment number and the second column lists the total number of kernels in that experiment. The third column enumerates the number of times, a second-best processor was chosen owing to the flexibility offered by the α value. And finally, the last column lists the different kernels for which the second-best processor was chosen and the frequency of this decision for that kernel. A key for the kernel names is as follows, nw: Needleman-Wunsch, bfs: Breadth First Search, srad: Speckle Reducing Anisotropic Diffusion, mi: Matrix Inverse, gem: Gaussian Electrostatic Model

Table 15. APT kernel allocation analyses for DFG Type-1 graphs.

α = 1.5			
Experiment no.	Total no. of kernels	Total different assignments	Kernel specific assignments
1	46	2	2-nw
2	58	0	0
3	50	0	0
4	73	0	0
5	69	0	0
6	81	0	0
7	125	0	0
8	93	0	0
9	132	0	0
10	157	0	0
α = 2			
Experiment no.	Total no. of kernels	Total different assignments	Kernel specific assignments

1	46	7	1-nw 6-bfs
2	58	0	0
3	50	0	0
4	73	1	1-bfs
5	69	0	0
6	81	0	0
7	125	2	2-bfs
8	93	0	0
9	132	0	0
10	157	0	0
α = 4			
Experiment no.	Total no. of kernels	Total different assignments	Kernel specific assignments
1	46	17	11-bfs 6-nw
2	58	17	11- nw 4- srad 1- mi 1- bfs
3	50	22	10- nw 1- srad 3- mi 8- bfs
4	73	13	3- nw 8- srad 1- mi 1- bfs
5	69	14	4- nw 8- srad 1- mi 1- bfs
6	81	24	11- nw 11- srad 1- mi 1- bfs

7	125	34	17- nw 10- srad 5- mi 2- bfs
8	93	36	21- nw 13- srad 2- mi
9	132	30	12- nw 10- srad 7- mi 1- bfs
10	157	47	14- nw 23- srad 10- mi
α = 8			
Experiment no.	Total no. of kernels	Total different assignments	Kernel specific assignments
1	46	17	11-bfs 6-nw
2	58	17	11- nw 4- srad 1- mi 1- bfs
3	50	25	14- nw 1- gem 10- mi
4	73	9	3- nw 4- srad 1- gem 1- bfs
5	69	17	8-srad 4-nw 4-mi 1-bfs
6	81	22	7-srad 11-nw 1-mi 1-bfs 2-gem

7	125	34	3-srad 17-nw 10-mi 2-bfs 2-gem
8	93	27	2-srad 21-nw 1-mi 3-gem
9	132	33	3-srad 12-nw 12-mi 1-bfs 5-gem
10	157	36	4-srad 14-nw 12-mi 6-gem
α = 16			
Experiment no.	Total no. of kernels	Total different assignments	Kernel specific assignments
1	46	18	1-cd 6-nw 11-bfs
2	58	17	4-srad 11-nw 1-mi 1-bfs
3	50	25	14-nw 10-mi 1-gem
4	73	10	4-srad 3-nw 1-mi 1-bfs 1-gem

5	69	18	8-srad 4-nw 5-mi 1-bfs
6	81	22	2-cd 2-srad 12-nw 1-mi 1-bfs 4-gem
7	125	35	3-srad 17-nw 11-mi 2-bfs 2-gem
8	93	30	2-srad 21-nw 4-mi 3-gem
9	132	37	3-srad 12-nw 16-mi 1-bfs 5-gem
10	157	43	6-srad 14-nw 18-mi 5-gem

The following table has the analyses of how different APT behaves for DFG Type-2.

Table 16. APT kernel allocation analyses for DFG Type-2 graphs.

α = 1.5			
Experiment no.	Total no. of kernels	Total different assignments	Kernel specific assignments
1	46	2	2-nw
2	58	0	0
3	50	0	0
4	73	0	0
5	69	0	0
6	81	0	0
7	125	0	0
8	93	0	0
9	132	0	0
10	157	0	0
α = 2			
Experiment no.	Total no. of kernels	Total different assignments	Kernel specific assignments
1	46	7	1-nw 6-bfs
2	58	0	0
3	50	0	0
4	73	1	1-bfs
5	69	0	0
6	81	0	0
7	125	2	2-bfs
8	93	0	0
9	132	0	0
10	157	0	0
α = 4			
Experiment no.	Total no. of kernels	Total different assignments	Kernel specific assignments
1	46	17	11-bfs 6-nw

2	58	17	11- nw 4- srad 1- mi 1- bfs
3	50	22	10- nw 1- srad 3- mi 8- bfs
4	73	13	3- nw 8- srad 1- mi 1- bfs
5	69	14	4- nw 8- srad 1- mi 1- bfs
6	81	24	11- nw 11- srad 1- mi 1- bfs
7	125	34	17- nw 10- srad 5- mi 2- bfs
8	93	36	21- nw 13- srad 2- mi
9	132	30	12- nw 10- srad 7- mi 1- bfs
10	157	47	14- nw 23- srad 10- mi
α = 8			
Experiment no.	Total no. of kernels	Total different assignments	Kernel specific assignments

1	46	17	11-bfs 6-nw
2	58	17	11- nw 4- srad 1- mi 1- bfs
3	50	25	14- nw 1- gem 10- mi
4	73	9	3- nw 4- srad 1- gem 1- bfs
5	69	17	8-srad 4-nw 4-mi 1-bfs
6	81	22	7-srad 11-nw 1-mi 1-bfs 2-gem
7	125	34	3-srad 17-nw 10-mi 2-bfs 2-gem
8	93	27	2-srad 21-nw 1-mi 3-gem
9	132	33	3-srad 12-nw 12-mi 1-bfs 5-gem
10	157	36	4-srad 14-nw 12-mi 6-gem

α = 16			
Experiment no.	Total no. of kernels	Total different assignments	Kernel specific assignments
1	46	18	1-cd 6-nw 11-bfs
2	58	17	4-srad 11-nw 1-mi 1-bfs
3	50	25	14-nw 10-mi 1-gem
4	73	10	4-srad 3-nw 1-mi 1-bfs 1-gem
5	69	18	8-srad 4-nw 5-mi 1-bfs
6	81	22	2-cd 2-srad 12-nw 1-mi 1-bfs 4-gem
7	125	35	3-srad 17-nw 11-mi 2-bfs 2-gem
8	93	30	2-srad 21-nw 4-mi 3-gem

9	132	37	3-srad 12-nw 16-mi 1-bfs 5-gem
10	157	43	6-srad 14-nw 18-mi 5-gem

α = 1.5			
Experiment no.	Total no. of kernels	Total different assignments	Kernel specific assignments
1	46	0	0
2	58	0	0
3	50	2	2-nw
4	73	0	0
5	69	0	0
6	81	0	0
7	125	0	0
8	93	1	1-nw
9	132	0	0
10	157	0	1-nw
α = 2			
Experiment no.	Total no. of kernels	Total different assignments	Kernel specific assignments
1	46	5	5-bfs
2	58	1	1-bfs
3	50	2	1-nw 1-bfs
4	73	1	1-bfs
5	69	0	0
6	81	2	2-bfs
7	125	2	2-bfs
8	93	1	1-nw
9	132	0	0
10	157	1	1-nw

α = 4			
Experiment no.	Total no. of kernels	Total different assignments	Kernel specific assignments
1	46	10	7-bfs 3-nw
2	58	13	1-bfs 4-srad 8-nw
3	50	13	6-bfs 1-mi 6-nw
4	73	15	6-srad 6-bfs 2-nw 1-mi
5	69	12	7-srad 1-mi 2-bfs 2-nw
6	81	24	9-srad 10-nw 4-bfs 1-mi
7	125	32	9-srad 16-nw 5-mi 2-bfs
8	93	32	9-srad 18-nw 2-mi 3-bfs
9	132	25	9-srad 10-nw 6-mi
10	157	41	19-srad 10-nw 9-mi 3-bfs

α = 8			
Experiment no.	Total no. of kernels	Total different assignments	Kernel specific assignments
1	46	10	3-nw 7-bfs
2	58	13	4-srad 8-nw 1-bfs
3	50	17	5-nw 5-mi 7-bfs
4	73	11	2-srad 2-nw 1-mi 3-gem 3-bfs
5	69	12	4-srad 2-nw 3-mi 2-bfs 1-gem
6	81	19	2-srad 11-nw 1-mi 2-bfs 3-gem
7	125	37	7-srad 16-nw 10-mi 2-bfs 2-gem
8	93	28	4-srad 18-nw 1-mi 2-gem 3-bfs
9	132	33	4-srad 12-nw 13-mi 4-gem

10	157	36	8-srad 10-nw 10-mi 3-bfs 5-gem
α = 16			
Experiment no.	Total no. of kernels	Total different assignments	Kernel specific assignments
1	46	10	3-nw 7-bfs
2	58	13	4-srad 8-nw 1-bfs
3	50	16	6-nw 5-mi 5-bfs
4	73	11	2-srad 2-nw 1-mi 3-gem 3-bfs
5	69	13	4-srad 2-nw 4-mi 2-bfs 1-gem
6	81	19	2-srad 11-nw 1-mi 2-bfs 3-gem
7	125	38	1-cd 7-srad 16-nw 10-mi 2-bfs 2-gem

8	93	29	4-srad 18-nw 2-mi 2-gem 3-bfs
9	132	34	3-srad 12-nw 14-mi 5-gem
10	157	43	1-cd 8-srad 10-nw 17-mi 2-bfs 5-gem

Bibliography

[1] G. Falcao, M. Owaida, D. Novo, M. Purnaprajna, N. Bellas, C. Antonopoulos, G. Karakonstantis, A. Burg, and P. Ienne. Shortening Design Time through Multiplatform Simulations with a Portable OpenCL Golden-model: The LDPC Decoder Case. *International Symposium on Field-Programmable Custom Computing Machines*, Apr. 2012.

[2] C. Fletcher, I. Lebedev, and N. Asadi. Bridging the GPGPU-FPGA efficiency gap. *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb. 2011.

[3] D. Llamocca, C. Carranza, and M. Pattichis. Separable FIR Filtering in FPGA and GPU Implementations: Energy, Performance, and Accuracy Considerations. *International Conference on Field Programmable Logic and Applications*, Sept. 2011.

[4] A.P.D. Binotto, D. Doering, T. Stetzelberger, P. McVittie, S. Zimmermann and C.E. Periera. A CPU, GPU, FPGA system for X-ray image processing using high-speed scientific cameras. SBAC-PAD, page 113-119. IEEE Computer Society, (2013)

[5] S. Skalicky, S. Lopez, M. Lukowiak. Distributed Execution of Transmural Electrophysiological Imaging with CPU, GPU, and FPGA. International Conference on ReConFigurable Computing and FPGAs. December 2013.

[6] Khokhar, A., Prasanna, V., Shaaban, M., Wang, C.L.: Heterogeneous Computing: Challenges and Opportunities. Computer 26(6) (June 1993)

[7] Chen, D., Singh, D.: Using OpenCL to Evaluate the Efficiency of CPUs, GPUs and FPGA for Information Filtering. International Conference on Field Programmable Logic and Applications (Aug. 2012)

[8] Skalicky, S., Lopez, S., Lukowiak, M., Letendre, J., Gasser, D.: Linear Algebra Computations in Heterogeneous Systems. IEEE International Conference on Application-specific Systems, Architectures and Processors (June 2013)

[9] Y.-K. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. 31(4), 1999.

[11] Liu, G.Q., Poh, K.L., Xie, M.: Iterative List Scheduling for Heterogeneous Computing. Journal of Parallel and Distributed Computing 65(5) (Jan. 2005)

[12] Cirou, B., Jeannot E. Triplet: A Clustering Scheduling Algorithm for Heterogeneous Systems. International Conference on Parallel Processing Workshops (2001)
[13] Boeres, C., Filho J.V., Rebello V.E.F. A Cluster-based Strategy for Scheduling Task on Heterogeneous Processors. Symposium on Computer Architecture and High Performance Computing (2004)
[14] Canon, L.C., Jeannot E., Sakellariou R., Zheng W. Comparative Evaluation of The Robustness of DAG Scheduling Heuristics. Grid Computing (2008)
[15] H. Arabnejad and J. Barbosa. List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table. *IEEE Transactions on Parallel and Distributed Systems*, PP(99), Mar. 2013.
[16] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3), 2002.
[17] C. Liu and S. Yang. A Heuristic Serial Schedule Algorithm for Unrelated Parallel Machine Scheduling with Precedence Constraints. *Journal of Software*, 6(6), June 2011.
[18] J. Wu, W. Shi, and B. Hong. Dynamic Kernel/Device Mapping Strategies for GPU-Assisted HPC Systems. *Job Scheduling Strategies for Parallel Processing*, May 2012.
[19] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 61(6), 2001.
[20] Llamocca, D., Carranza, C., Pattichis, M.: Separable FIR Filtering in FPGA and GPU Implementations: Energy, Performance, and Accuracy Considerations. International Conference on Field Programmable Logic and Applications (September 2011)
[21] Fletcher, C., Lebedev, I., Asadi, N.: Bridging the GPGPU-FPGA Efficiency Gap. ACM/SIGDA International Symposium on Field Programmable Gate Arrays (Mar. 2011)
[22] Yang, D., Sun, J., Lee, J.: Performance Comparison of Cholesky Decomposition on GPUs and FPGAs. Symposium on Application Accelerators in High-Performance Computing (July 2010)

[23] Skalicky, S., Lopez, S., Lukowiak, M., Letendre, J., Gasser, D.: Linear Algebra Computations in Heterogeneous Systems. IEEE International Conference on Application-specific Systems, Architectures and Processors (June 2013)
[24] K. Krommydas, W. Feng, M. Owaida, C. Antonopoulos and N. Bellas, "On the characterization of opencl dwarfs on fixed and reconfigurable platforms", Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on, pp. 153-160
[25] H. Meuer, E. Strohmaier, J. Dongarra, H. Simon, "Top 500 supercomputers", 2011, http://www.top500.org
[26] J. Jackson, "Supercomputing top500 brews discontent," 2010, http://www.pcworld.idg.com.au/article/368598/supercomputing_top500_brews_discontent/.
[27] M. Showerman, J. Enos, A. Pant et al., "QP: a heterogeneous multi-accelerator cluster," in Proceedings of the 10th LCI International Conference on High-Performance Cluster Computing, vol. 7800, pp. 1–8, Boulder, Colo, USA, 2009.
[28] K. H. Tsoi and W. Luk, "Axel: a heterogeneous cluster with FPGAs and GPUs," in Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA '01), pp. 115–124, Monterey, Calif, USA, 2010.
[29] Ra Inta, David J. Bowman, and Susan M. Scott, "The "Chimera": An Off-The-Shelf CPU/GPGPU/FPGA Hybrid Computing Platform," International Journal of Reconfigurable Computing, vol. 2012, Article ID 241439, 10 pages, 2012.
[30] S. Huang, S. Xiao and W. Feng, "On the energy efficiency of graphics processing units for scientific computing," *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, Rome, 2009, pp. 1-8.
[31] P. Colella, "Defining Software Requirements for Scientific Computing," presentation, 2004
[32] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Dec. 2006

[33] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co. Inc., 1995.

[34] R. Puigjaner. Performance Modeling of Computer Networks. *IFIP/ACM Latin America Conference on Towards a Latin American Agenda for Network Research*, Oct. 2003.

[35] J. Herrmann, J. M. Proth, and N. Sauer. Heuristics for Unrelated Machine Scheduling with Precedence Constraints. *European Journal of Operational Research*, 102(3), 1997.

[36] Y. Gong, M. E. Pierce, and G. C. Fox. Dynamic Resource-Critical Workflow Scheduling in Heterogeneous Environments. *Job Scheduling Strategies for Parallel Processing*, May 2009.

[37] Khokhar, A., Prasanna, V., Shaaban, M., Wang, C.L.: Heterogeneous Computing: Challenges and Opportunities. Computer 26(6) (June 1993)

[38] Needleman, S., and Wunsch, C., 1970, A general method applicable to the search for similarities in the amino acid sequence of two proteins: J. Mol. Biol., 48,443-453.

[39] Commandant Benoit, Note sur une méthode de résolution des équations normales provenant de l'application de la méthode des moindres carrés à un système d'équations linéaires en nombre inférieur à celui des inconnues (Procédé du Commandant Cholesky), Bulletin Géodésique 2 (1924), 67-77